A Hardware Implementation of the MCAS Synchronization Primitive
Appendix

1 Optimized Implementation: MCAS-OPT

In this section, we improve upon our base implementation MCAS-BASE with an optimized implementation MCAS-OPT. In the former, the `MTS` instructions to populate the MCAS Table were executed just before the `MCAS` instruction. However, analyzing common concurrent data structures, we observed that the addresses of the MCAS-related memory locations are known much earlier in the program. Thus, at runtime, it is possible to make the MCAS Table entry much earlier. This is extremely useful because in the time interval between when the entry was made and the `MCAS` instruction, through tracking of the cache coherence messages, we can know if the concerned memory location was written to by some other core. If so, then it is highly likely that this MCAS will fail when executed. We can thus perform an *"early back-off"* and avoid executing the Acquire Lock and Compare Phase.

1.1 Placement of `MTS` Instructions

We design a compiler pass that statically analyzes the program and places `MTS` instructions as *high* up in the program as possible, while maintaining functional correctness. The proposed method is a Worklist algorithm for iterative forward data-flow analysis. The function containing the MCAS invocation is considered as a graph of basic blocks $G = \langle N, E \rangle$. Let the maximum arity of the MCAS primitive be k. Each basic block B is associated with two vectors in_B and out_B, each of size $k - in_B$ represents the data-flow information reaching the entry to B, while out_B represents the data-flow information leaving B. The concerned data-flow information consists of the instruction points (IP) values (one for each MCAS memory location)‘ of the earliest instruction after which it is safe to place the `MTS` instruction for that address. Algorithm 1 shows the working of the pass.

The operator $\sqcap$ essentially combines the output data-flows of predecessor basic blocks to give the input data-flow of the current block. Let A and B be the output flows of two predecessor blocks. For any i, let a and b be the i^{th}

elements of A and B respectively. $headIP$ is the IP of the first instruction of the current basic block. $\sqcap$ is defined as follows:

$$\sqcap(a,b) = \begin{cases} -1 & \text{if } a = b = -1 \\ a & \text{if } a \neq -1, b = -1 \\ b & \text{if } b \neq -1, a = -1 \\ \text{a} & \text{if } a = b, a \neq -1, b \neq -1 \\ \text{headIP} & \text{if } a \neq b, a \neq -1, b \neq -1 \end{cases}$$

The local analysis within a basic block is done by the flow function $F()$. It updates the resultant vector if any new definition of the address field is encountered in that basic block.

Input: N: set of basic blocks
$entry$: the entry basic block of the function
$exit$: the basic block containing the MCAS invocation
k: the maximum arity of an MCAS instruction
Output: $result$: vector of IP addresses of length k
Variables: in: set of vectors of IP addresses of length k, one vector per basic block
B: basic block
$worklist$: set of basic blocks
$effect$, $totaleffect$: vectors of IP addresses of length k

```
worklist = N
for each B ϵ N do
    initialize vector in_B to -1
end
repeat
    B = first node of worklist
    worklist = worklist − B
    totaleffect: initialize vector to -1
    for each P ϵ Predecessors(B) do
        effect = F(in_B)
        totaleffect = totaleffect ⊓ effect
    end
    if in_B ≠ totaleffect then
        in_B = totaleffect
        worklist = worklist ∪ successors(B)
    end
until worklist = ϕ
result = in_exit
return result
```

Algorithm 1: Worklist_Iterate($N, entry, exit, k$)

1.2 Early Back-Off

The compiler algorithm helps us place `MTS` instructions as soon as the address is unambiguously known. However, at this point, the old value and new value may not be known. The `MTS` instruction therefore sets these to -1 (an arbitrary value deemed illegal). As in MCAS-BASE, just before the MCAS instruction, when all parameters of the MCAS have been resolved, `MTS` instructions are again placed by the compiler. These *later* `MTS`s contain all legal values. The MCAS Table is augmented with an extra 1-bit flag *MCAS Invalid Flag* (MIF) (1 flag per MCAS Table). When a cache receives an *invalidate* message from the directory, due to some other core writing to the same line, the former cache checks its MCAS Table to see if it has an entry corresponding to the same address. If it does, it sets the MCAS Invalid Flag (MIF) to 1. Now, at any point in time when the Lock Acquire and Compare Phase is in progress, if the MIF is found to be 1, the phase is aborted. The MCAS is deemed to be failed, and the Exit Phase is executed.

MCAS-OPT was implemented and synthesized. The hardware overhead increased marginally, amounting to 0.0466%.

2 Benchmarks

The concurrent data structures used for our evaluation are: Binary Search Tree, Sorted List, Doubly Linked List, HashSet, Queue and Stack. There are two reasons for choosing this set of benchmarks. Firstly, these data structures are widely used in many real-world applications, and hence provide credibility to the proposal. Secondly, each of these benchmarks requires MCASs of different arity. This helps us gain a better insight into the behavior of a hardware implementation of such a primitive. Each benchmark is composed of 32 threads. Each thread makes a total of 300 operations (empirically found to reach steady state) on the shared data structure – alternate insertions and deletions of random elements to the data structure. We use three versions of benchmarks (most optimal implementations known in literature [1] [2]): with locks, lock-free, and with MCAS. The benchmarks are implemented in C++. They have been compiled with GCC version 4.7.2, with the -std=c++0x option to enable C++11 support.

	Lock-based	**CAS-based Lock-free**	**MCAS-based Lock-free**
Binary Search Tree	111	320	162
Doubly Linked List	53	171	51
Hashset	87	245	85
Ordered List	77	101	83
Queue	40	66	36
Stack	33	46	46
Average	66.83	158.17	77.00

Table 1: Lock-based v/s CAS-based Lock-free v/s MCAS-based Lock-free : Lines of code comparison

Table 1 shows a comparison between the different implementations in terms of lines of code. Lines of code is a popular metric to compare the ease of programming, debugging and maintaining software. As is clearly evident from the statistics, the availability of an MCAS primitive greatly simplifies the task of a programmer.

2.1 Lock Implementation

Before each operation, the thread has to acquire a lock on the structure, which it releases once it completes performing the operation.

2.1.1 Binary Search Tree (BST)

Each thread in the benchmark performs alternate insertions and deletions in the binary search tree. The tree is initially populated with 15 elements. Each thread performs 300 operations.

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
struct bst {
  int data;
  struct bst *left;
  struct bst *right;
};
typedef struct bst bst_t;

bst_t *get_new_node(int val) {
  bst_t *node = (bst_t *) malloc(sizeof(bst_t));
  node->data = val;
  node->left = NULL;
  node->right = NULL;
  return node;
}
bst_t *root = NULL;

bst_t *insert(bst_t *root, int val) {
  if (!root)
    return get_new_node(val);
  bst_t *prev = NULL, *ptr = root;
  char type = ' ';
  while (ptr) {
    prev = ptr;
    if (val < ptr->data) {
      ptr = ptr->left;
      type = 'l';
    } else {
      ptr = ptr->right;
      type = 'r';
    }
  }
  if (type == 'l')
    prev->left = get_new_node(val);
  else
    prev->right = get_new_node(val);
  return root;
}

int find_minimum_value(bst_t *ptr) {
  int min = ptr ? ptr->data : 0;
  while (ptr) {
    if (ptr->data < min)
      min = ptr->data;
    if (ptr->left) {
```

```
            ptr = ptr->left;
        } else if (ptr->right) {
            ptr = ptr->right;
        } else
            ptr = NULL;
    }
    return min;
}

bst_t *deleter(bst_t *root, int val) {
    bst_t *prev = NULL, *ptr = root;
    char type = ' ';
    while (ptr) {
        if (ptr->data == val) {

            if (!ptr->left && !ptr->right) { // node to be removed has no children's
                if (ptr != root && prev) { // delete leaf node
                    if (type == 'l')
                        prev->left = NULL;
                    else
                        prev->right = NULL;
                } else
                    root = NULL; // deleted node is root
            } else if (ptr->left && ptr->right) { // node to be removed has two
                children's
                ptr->data = find_minimum_value(ptr->right); // find minimum value
                    from right subtree
                val = ptr->data;
                prev = ptr;
                ptr = ptr->right; // continue from right subtree delete min node
                type = 'r';
                continue;
            } else { // node to be removed has one children
                if (ptr == root) { // root with one child
                    root = root->left ? root->left : root->right;
                } else { // subtree with one child
                    if (type == 'l')
                        prev->left = ptr->left ? ptr->left : ptr->right;
                    else
                        prev->right = ptr->left ? ptr->left : ptr->right;
                }
            }
        }
        prev = ptr;
        if (val < ptr->data) {
            ptr = ptr->left;
            type = 'l';
            height++;
        } else {
            ptr = ptr->right;
            type = 'r';
            height++;
        }
    }
    return root;
}
```

```c
void enq(int value) {
   pthread_mutex_lock(&mutex1);
   root = insert(root, value);
   pthread_mutex_unlock(&mutex1);
}

void deq(int value) {
   pthread_mutex_lock(&mutex1);
   root = deleter(root, value);
   pthread_mutex_unlock(&mutex1);
}
```

2.1.2 Doubly Linked List

```c
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
struct node {
   int data;
   struct node *next;
   struct node *prev;
};
struct node* head = NULL;

void insertAfter(struct node* prev_node, int new_data) {
   if (prev_node == NULL)
   {
      printf("the given previous node cannot be NULL");
      return;
   }
   struct node* new_node = (struct node*) malloc(sizeof(struct node));
   new_node->data = new_data;
   new_node->next = prev_node->next;
   prev_node->next = new_node;
   new_node->prev = prev_node;
   if (new_node->next != NULL)
      new_node->next->prev = new_node;
}

void insert(int key) {
   pthread_mutex_lock(&mutex1);
   struct node *prev = head, *curr = prev;
   while (curr->data < key) {
      prev = curr;
      curr = curr->next;
   }
   insertAfter(prev, key);
   pthread_mutex_unlock(&mutex1);
}

void deleteNode(struct node *del) {
   if (del == NULL)
      return;
   del->next->prev = del->prev;
   del->prev->next = del->next;
   del = NULL;
   free(del);
}

void deleten(int key) {
   pthread_mutex_lock(&mutex1);
   struct node *prev = head, *curr = prev;
   while (curr->data < key) {
      prev = curr;
      curr = curr->next;
   }
   deleteNode(curr);
   pthread_mutex_unlock(&mutex1);
}
```

2.1.3 Hash Set

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
static vector<struct node*> table;
struct node {
   int data;
   struct node *next;
};

void linkadd(int num, int index) {
   struct node *temp;
   temp = (struct node *) malloc(sizeof(struct node));
   temp->data = num;
   if (table[index] == NULL)
   {
      table[index] = temp;
      table[index]->next = NULL;
   } else {
      temp->next = table[index];
      table[index] = temp;
   }
}

bool linkdelete(int num, int index) {
   struct node *temp, *prev;
   temp = table[index];
   if (temp != NULL)
   {
      table[index] = temp->next;
      return true;
   }
   return false;
}

bool linkcontain(int num, int index) {
   if (table[index] == NULL)
   {
      return false;
   }
   struct node* r = table[index];
   while (r != NULL) {
      if (r->data == num)
         return true;
      r = r->next;
   }
   return false;
}

class CoarseHashSet {
   int size;
public:
   CoarseHashSet(int capacity) {
      size = 0;
      for (int i = 0; i < capacity; i++) {
         struct node *temp = NULL;
         table.push_back(temp);
      }
```

```
    }

    bool contains(int x) {
        std::tr1::hash<int> hash_fn;
        int a = hash_fn(x);
        int myBucket = abs(a % table.size());
        bool ans = linkcontain(x, myBucket);
        return ans;
    }

    void add(int x) {
        pthread_mutex_lock(&mutex1);
        std::tr1::hash<int> hash_fn;
        int a = hash_fn(x);
        int myBucket = abs(a % table.size());
        linkadd(x, myBucket);
        size = size + 1;
        pthread_mutex_unlock(&mutex1);
        if (size / table.size() > 4)
            resize();
    }

    void remove(int x) {
        pthread_mutex_lock(&mutex1);
        std::tr1::hash<int> hash_fn;
        int a = hash_fn(x);
        int myBucket = abs(a % table.size());
        bool result = linkdelete(x, myBucket);
        size = result ? size - 1 : size;
        pthread_mutex_unlock(&mutex1);
    }
};
```

2.1.4 List

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

class LockedList {

public:
   class Node {
   public:
      int item;
      int key;
      Node *next;
      Node(int item) {
         this->item = item;
         this->key = item;
      }
   };
   Node *head;
   Node *tail;

   LockedList() {
      // Add sentinels to start and end
      head = new Node(INT_MIN);
      tail = new Node(INT_MAX);
      head->next = this->tail;
   }

   bool add(int item) {
      Node *pred, *curr;
      int key = item;
      bool flag;

      pthread_mutex_lock(&mutex1);
      pred = head;
      curr = pred->next;
      while (curr->key < key) {
         pred = curr;
         curr = curr->next;
      }

      Node *node = new Node(item);
      node->next = curr;
      pred->next = node;
      flag = true;
      pthread_mutex_unlock(&mutex1);
      return flag;
   }

   bool remove(int item) {
      Node *pred, *curr;
      int key = item;
      bool flag;
      pthread_mutex_lock(&mutex1);
      pred = this->head;
      curr = pred->next;
      while (curr->key < key) {
         pred = curr;
```

```
            curr = curr->next;
        }
        pred->next = curr->next;
        flag = true;
        pthread_mutex_unlock(&mutex1);
        return flag;
    }

    bool contains(int item) {
        Node *pred, *curr;
        int key = item;
        pthread_mutex_lock(&mutex1);
        pred = head;
        curr = pred->next;
        while (curr->key < key) {
            pred = curr;
            curr = curr->next;
        }
        pthread_mutex_unlock(&mutex1);
        return (key == curr->key);
    }
};
```

2.1.5 Queue

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
struct qNode {
   int value;
   struct qNode *next;
};

struct qNode *front = NULL, *rear = NULL;

void enq(int item) {
   pthread_mutex_lock(&mutex1);
   struct qNode *newN = new struct qNode;
   newN->value = item;
   newN->next = NULL;
   if (front == NULL && rear == NULL) {
      front = newN;
      rear = newN;
   } else {
      rear->next = newN;
      rear = newN;
   }
   pthread_mutex_unlock(&mutex1);
}

int deq() {
   int t;
   pthread_mutex_lock(&mutex1);
   if (front != NULL) {
      struct qNode *temp = front;
      t = temp->value;
      if (front == rear) {
         front = NULL;
         rear = NULL;
      } else
         front = front->next;
   } else {
      cout << "queue is empty" << endl;
   }
   pthread_mutex_unlock(&mutex1);
   return t;
}
```

2.1.6 Stack

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
class stack {
public:
   class Node {
   public:
      int item;
      Node *next;
      Node(int item) {
         this->item = item;
         this->next = NULL;
      }
   };
   Node *top;

   void push(int value) {
      Node *node = new Node(value);
      pthread_mutex_lock(&mutex1);
      node->next = top;
      top = node;
      pthread_mutex_unlock(&mutex1);

   }

   void pop() {
      int t;
      pthread_mutex_lock(&mutex1);
      if (top != NULL) {
         t = top->item;
         top = top->next;
      }
      pthread_mutex_unlock(&mutex1);
   }
};
```

2.2 CAS-based Lock-free Implementation

Library Used by all Benchmarks

```
static size_t compareAndExchange( volatile size_t* addr, size_t oldval, size_t
    newval ){
  size_t ret;
  __asm__ volatile( "lock cmpxchg %2, %1\n\t":"=a"(ret), "+m"(*addr):
      "r"(newval), "0"(oldval): "memory" );
  return ret;
}

static size_t compareAndExchangeTid( volatile size_t* addr, int oldval, int
    newval ){
  int ret;
  __asm__ volatile( "lock cmpxchg %2, %1\n\t":"=a"(ret), "+m"(*addr):
      "r"(newval), "0"(oldval): "memory" );
  return ret;
}
static size_t AtomicExchange(volatile size_t* ptr, size_t new_value) {
 __asm__ __volatile__("xchg %1,%0": "=r" (new_value): "m" (*ptr), "0"
     (new_value): "memory");
 return new_value; // Now it's the previous value.
}

template<typename T,unsigned N=sizeof (uint32_t)>
struct DPointer {
public:
  union {
    uint64_t ui;
    struct {
      T* ptr;
      size_t mark;
    };
  };

  DPointer() : ptr(NULL), mark(0) {}
  DPointer(T* p) : ptr(p), mark(0) {}
  DPointer(T* p, size_t c) : ptr(p), mark(c) {}

  bool cas(DPointer<T,N> const& nval, DPointer<T,N> const & cmp)
  {
    bool result;
    __asm__ __volatile__(
      "lock cmpxchg8b %1\n\t"
      "setz %0\n"
      : "=q" (result)
       ,"+m" (ui)
      : "a" (cmp.ptr), "d" (cmp.mark)
       ,"b" (nval.ptr), "c" (nval.mark)
      : "cc"
    );
    return result;
  }

  // We need == to work properly
  bool operator==(DPointer<T,N> const&x) { return x.ui == ui; }
```

```
};

template<typename T>
struct DPointer <T,sizeof (uint64_t)> {
public:
   union {
      uint64_t ui[2];
      struct {
         T* ptr;
         size_t mark;
      } __attribute__ (( __aligned__( 16 ) ));
   };

   DPointer() : ptr(NULL), mark(0) {}
   DPointer(T* p) : ptr(p), mark(0) {}
   DPointer(T* p, size_t c) : ptr(p), mark(c) {}

   bool cas(DPointer<T,8> const& nval, DPointer<T,8> const& cmp)
   {
      bool result;
      __asm__ __volatile__ (
         "lock cmpxchg16b %1\n\t"
         "setz %0\n"
         : "=q" ( result )
          ,"+m" ( ui )
         : "a" ( cmp.ptr ), "d" ( cmp.mark )
          ,"b" ( nval.ptr ), "c" ( nval.mark )
         : "cc"
      );
      return result;
   }

   // We need == to work properly
   bool operator==(DPointer<T,8> const&x) { return x.ptr == ptr && x.mark ==
        mark; }
};
```

2.2.1 Binary Search Tree

```
class Node {
public:

   long key;
   long value;
   DPointer<Node, sizeof(size_t)> lChild;
   DPointer<Node, sizeof(size_t)> rChild;
   Node() {
   }

   Node(long key, long value) {
      this->key = key;
      this->value = value;
   }

   Node(long key, long value, DPointer<Node, sizeof(size_t)> lChild
        , DPointer<Node, sizeof(size_t)> rChild) {
      this->key = key;
      this->value = value;
      this->lChild = lChild;
      this->rChild = rChild;
   }
};

class SeekRecord {
public:
   Node *ancestor;
   Node *successor;
   Node *parent;
   Node *leaf;

   SeekRecord() {
   }
   SeekRecord(Node *ancestor, Node *successor, Node *parent, Node *leaf) {
      this->ancestor = ancestor;
      this->successor = successor;
      this->parent = parent;
      this->leaf = leaf;
   }
};

class LockFreeBST {
public:
   static Node *grandParentHead;
   static Node *parentHead;
   static LockFreeBST *obj;

   LockFreeBST() {
      createHeadNodes();
   }

   long lookup(long target) {
      Node *node = grandParentHead;
      while (node->lChild.ptr != NULL) //loop until a leaf or dummy node is
           reached
```

```
    {
      if (target < node->key) {
        node = node->lChild.ptr;
      } else {
        node = node->rChild.ptr;
      }
    }

    if (target == node->key)
      return (1);
    else
      return (0);
  }

  void add(long insertKey) {
    int nthChild;
    Node *node;
    Node *pnode;
    SeekRecord *s;
    while (true) {
      nthChild = -1;
      pnode = parentHead;
      node = parentHead->lChild.ptr;
      while (node->lChild.ptr != NULL) //loop until a leaf or dummy node is reached
      {
        if (insertKey < node->key) {
          pnode = node;
          node = node->lChild.ptr;
        } else {
          pnode = node;
          node = node->rChild.ptr;
        }
      }

      Node *oldChild = node;

      if (insertKey < pnode->key) {
        nthChild = 0;
      } else {
        nthChild = 1;
      }

      //leaf node is reached
      if (node->key == insertKey) {
        //key is already present in tree. So return
        return;
      }

      Node *internalNode, *lLeafNode, *rLeafNode;
      if (node->key < insertKey) {
        rLeafNode = new Node(insertKey, insertKey);
        internalNode = new Node(insertKey, insertKey, DPointer<Node, sizeof(size_t)>(node, 0), DPointer<Node, sizeof(size_t)>(rLeafNode, 0));
      } else {
        lLeafNode = new Node(insertKey, insertKey);
```

```
            internalNode = new Node(node->key, node->key, DPointer<Node, sizeof(size_t)>(lLeafNode, 0), DPointer<Node, sizeof(size_t)>(node, 0));
         }

         if (nthChild == 0) {
            if (pnode->lChild.cas(DPointer<Node, sizeof(size_t)>(internalNode, 0), oldChild)) {
               return;
            } else {
               //insert failed; help the conflicting delete operation
               if (node == pnode->lChild.ptr) { // address has not changed. So CAS would have failed coz of flag/mark only
                  //help other thread with cleanup
                  s = seek(insertKey);
                  cleanUp(insertKey, s);
               }
            }
         } else {
            if (pnode->rChild.cas(DPointer<Node, sizeof(size_t)>(internalNode, 0), DPointer<Node, sizeof(size_t)>(oldChild, 0))) {
               return;
            } else {
               if (node == pnode->rChild.ptr) {
                  s = seek(insertKey);
                  cleanUp(insertKey, s);
               }
            }
         }
      }
   }

   void remove(long deleteKey) {
      bool isCleanUp = false;
      SeekRecord *s;
      Node *parent;
      Node *leaf = NULL;
      while (true) {
         s = seek(deleteKey);
         if (!isCleanUp) {
            leaf = s->leaf;
            if (leaf->key != deleteKey) {
               return;
            } else {
               parent = s->parent;
               if (deleteKey < parent->key) {
                  if (parent->lChild.cas(DPointer<Node, sizeof(size_t)>(leaf, 2), leaf)) {
                     isCleanUp = true;
                     //do cleanup
                     if (cleanUp(deleteKey, s)) {
                        return;
                     }
                  } else {
                     if (leaf == parent->lChild.ptr) {
                        cleanUp(deleteKey, s);
                     }
```

```
                }
            } else {
                if (parent->rChild.cas(DPointer<Node, sizeof(size_t)>(leaf, 2),
                    leaf)) {
                    isCleanUp = true;
                    //do cleanup
                    if (cleanUp(deleteKey, s)) {
                        return;
                    }
                } else {
                    if (leaf == parent->rChild.ptr) {
                        //help other thread with cleanup
                        cleanUp(deleteKey, s);
                    }
                }
            }
        }
    } else {
        if (s->leaf == leaf) {
            //do cleanup
            if (cleanUp(deleteKey, s)) {
                return;
            }
        } else {
            //someone helped with my cleanup. So I'm done
            return;
        }
    }
  }
}

int setTag(int stamp) {
  switch (stamp) // set only tag
  {
  case 0:
    stamp = 1; // 00 to 01
    break;
  case 2:
    stamp = 3; // 10 to 11
    break;
  }
  return stamp;
}

int copyFlag(int stamp) {
  switch (stamp) //copy only the flag
  {
  case 1:
    stamp = 0; // 01 to 00
    break;
  case 3:
    stamp = 2; // 11 to 10
    break;
  }
  return stamp;
}
```

```
  bool cleanUp(long key, SeekRecord *s) {
    Node *ancestor = s->ancestor;
    Node *parent = s->parent;
    Node *oldSuccessor;
    size_t oldStamp;
    Node *sibling;
    size_t siblingStamp;

    if (key < parent->key) { //xl case
      if (parent->lChild.mark > 1) { // check if parent to leaf edge is already flagged. 10 or 11
        //leaf node is flagged for deletion. tag the sibling edge to prevent any modification at this edge now
        sibling = parent->rChild.ptr;
        siblingStamp = parent->rChild.mark;
        siblingStamp = setTag(siblingStamp); // set only tag

        parent->rChild.cas(DPointer<Node, sizeof(size_t)>(sibling, siblingStamp), sibling);
        sibling = parent->rChild.ptr;
        siblingStamp = parent->rChild.mark;
      } else {
        //leaf node is not flagged. So sibling node must have been flagged for deletion
        sibling = parent->lChild.ptr;
        siblingStamp = parent->lChild.mark;
        siblingStamp = setTag(siblingStamp); // set only tag

        parent->lChild.cas(DPointer<Node, sizeof(size_t)>(sibling, siblingStamp), sibling);
        sibling = parent->lChild.ptr;
        siblingStamp = parent->lChild.mark;
      }
    } else { //xr case
      if (parent->rChild.mark > 1) { // check if parent to leaf edge is already flagged. 10 or 11
        //leaf node is flagged for deletion. tag the sibling edge to prevent any modification at this edge now
        sibling = parent->lChild.ptr;
        siblingStamp = parent->lChild.mark;
        siblingStamp = setTag(siblingStamp); // set only tag

        parent->lChild.cas(DPointer<Node, sizeof(size_t)>(sibling, siblingStamp), sibling);
        sibling = parent->lChild.ptr;
        siblingStamp = parent->lChild.mark;
      } else {
        //leaf node is not flagged. So sibling node must have been flagged for deletion
        sibling = parent->rChild.ptr;
        siblingStamp = parent->rChild.mark;
        siblingStamp = setTag(siblingStamp); // set only tag

        parent->rChild.cas(DPointer<Node, sizeof(size_t)>(sibling, siblingStamp), sibling);
        sibling = parent->rChild.ptr;
        siblingStamp = parent->rChild.mark;
```

```
      }
    }

    if (key < ancestor->key) {
      siblingStamp = copyFlag(siblingStamp); //copy only the flag
      oldSuccessor = ancestor->lChild.ptr;
      oldStamp = ancestor->lChild.mark;
      return (ancestor->lChild.cas(DPointer<Node, sizeof(size_t)>(sibling,
           siblingStamp), DPointer<Node, sizeof(size_t)>(oldSuccessor,
           oldStamp)));
    } else {
      siblingStamp = copyFlag(siblingStamp); //copy only the flag
      oldSuccessor = ancestor->rChild.ptr;
      oldStamp = ancestor->rChild.mark;
      return (ancestor->rChild.cas(DPointer<Node, sizeof(size_t)>(sibling,
           siblingStamp), DPointer<Node, sizeof(size_t)>(oldSuccessor,
           oldStamp)));
    }
  }

  SeekRecord* seek(long key) {
    DPointer<Node, sizeof(size_t)> parentField;
    DPointer<Node, sizeof(size_t)> currentField;
    Node *current;

    //initialize the seek record
    SeekRecord *s = new SeekRecord(grandParentHead, parentHead, parentHead,
         parentHead->lChild.ptr);

    parentField = s->ancestor->lChild;
    currentField = s->successor->lChild;

    while (currentField.ptr != NULL) {
      current = currentField.ptr;
      //move down the tree
      //check if the edge from the current parent node in the access path is
           tagged
      if (parentField.mark == 0 || parentField.mark == 2) { // 00, 10 untagged
        s->ancestor = s->parent;
        s->successor = s->leaf;
      }
      //advance parent and leaf pointers
      s->parent = s->leaf;
      s->leaf = current;
      parentField = currentField;
      if (key < current->key) {
        currentField = current->lChild;
      } else {
        currentField = current->rChild;
      }
    }
    return s;
  }

  void createHeadNodes() {
    long key = LONG_MAX;
    long value = LONG_MIN;
```

```
    parentHead = new Node(key, value, DPointer<Node, sizeof(size_t)>(new
        Node(key, value), 0), DPointer<Node, sizeof(size_t)>(new Node(key,
        value), 0));
    grandParentHead = new Node(key, value, DPointer<Node,
        sizeof(size_t)>(parentHead, 0), DPointer<Node, sizeof(size_t)>(new
        Node(key, value), 0));
  }
};
Node * LockFreeBST::grandParentHead = NULL;
Node * LockFreeBST::parentHead = NULL;
LockFreeBST *LockFreeBST::obj = NULL;
```

2.2.2 Doubly Linked List

```
class doublylinked {
public:
   class Node {
   public:
      int value;
      DPointer<doublylinked::Node, sizeof(size_t)> after;
      Node *before;

      Node() {
      }
      Node(int key) {
         this->value = key;
         this->before = NULL;
         this->after = DPointer<doublylinked::Node, sizeof(size_t)>();
      }
   };
   Node *headdummy, *taildummy;

   bool deleten(int key) {
      Node *pred = headdummy, *curr;
      curr = pred->after.ptr;
      while (curr->value < key) {
         pred = curr;
         curr = curr->after.ptr;
      }
      return deleteNode(curr, true);
   }

   bool deleteNode(Node *thisNode, bool retry) {
      DPointer<doublylinked::Node, sizeof(size_t)> nextref;
      while (true) {
         nextref = thisNode->after;
         if (nextref.mark)
            return false;
         Node *next = nextref.ptr;
         if (thisNode->after.cas(DPointer<Node, sizeof(size_t)>(next, true),
             next));
         break;
         if (!retry)
            return false;
      }
      getBack(thisNode);
      return true;
   }

   Node* getBack(Node *refNode) {
      Node *prevref = refNode->before;
      Node *currentNode = refNode;
      while (true) {
         prevref = currentNode->before;
         Node *backnode = prevref;
         DPointer<doublylinked::Node, sizeof(size_t)> backAtref = backnode->after;
         Node *backAftNode = backAtref.ptr;
         if (backAtref.mark)
            currentNode = backnode;
```

```
      else if (backAftNode == refNode)
        return backnode;
      else {
        Node *maybeback = fixforwarduntil(backnode, refNode);
        if ((maybeback == NULL) && (backnode->after.mark))
          currentNode = backnode;
        else
          return maybeback;
      }
    }
  }

  Node* fixforwarduntil(Node *thisNode, Node *laterNode) {
    Node *nextnode, *worknode = thisNode;
    DPointer<doublylinked::Node, sizeof(size_t)> thisnodeAtref, workNodeAtref,
        laternodeAtref;

    while (true) {
      thisnodeAtref = thisNode->after;
      if (thisnodeAtref.mark)
        return NULL;
      laternodeAtref = laterNode->after;
      if ((laternodeAtref.ptr != NULL) && (laternodeAtref.mark))
        return NULL;
      workNodeAtref = worknode->after;
      if (workNodeAtref.ptr == NULL)
        return NULL;
      if (!(workNodeAtref.mark)) {
        fixforward(worknode);
        workNodeAtref = worknode->after;
      }
      nextnode = workNodeAtref.ptr;
      if (nextnode == laterNode)
        return worknode;
      else if (nextnode->after == NULL
        ) return NULL;
      else
        worknode = nextnode;
    }
  }

  void initialise() {
    Node *a = new Node(0);
    headdummy = a;
    taildummy = a;
    Node *b = new Node(100000000);
    b->before = a;
    a->after = DPointer < doublylinked::Node, sizeof(size_t) > (b, 0);
  }

  bool add(int key) {
    Node *mynode = new Node(key);
    Node *pred = headdummy, *curr;
    curr = pred->after.ptr;
    while (curr->value < key) {
      if (pred == curr)
        break;
```

```
            pred = curr;
            curr = curr->after.ptr;
        }
        if (headdummy == pred)
            return insertafter(headdummy, mynode);
        else
            insertafter(pred, mynode);
    }

    bool insertafter(Node *previous, Node *mynode) {
        while (true) {
            DPointer<doublylinked::Node, sizeof(size_t)> prevAtref = previous->after;
            if (prevAtref.mark)
                return false;
            Node *prevafter = fixforward(previous);
            if (insertBetween(mynode, previous, prevafter))
                return true;
        }
    }

    bool insertBetween(Node *thisNode, Node *prev, Node *after) {
        thisNode->before = prev;
        thisNode->after = DPointer < doublylinked::Node, sizeof(size_t) > (after,
             0);
        if (prev->after.cas(DPointer<Node, sizeof(size_t)>(thisNode, false),
             after)) {
            reflectforward(thisNode);
            return true;
        }
        return false;
    }

    Node* fixforward(Node *thisNode) {
        DPointer<doublylinked::Node, sizeof(size_t)> thisAtref = thisNode->after;
        Node *laterNode = thisAtref.ptr;
        Node *laterLater;
        while (true) {
            DPointer<doublylinked::Node, sizeof(size_t)> nextref = laterNode->after;
            if (nextref == NULL || !nextref.mark) {
                reflectforward(thisNode);
                return laterNode;
            } else {
                laterLater = nextref.ptr;
                thisNode->after.cas(DPointer<Node, sizeof(size_t)>(laterLater,
                     false), laterNode);
                laterNode = laterLater;
            }
        }
    }

    void reflectforward(Node *previous) {
        DPointer<doublylinked::Node, sizeof(size_t)> prevAtref = previous->after;
        if (prevAtref.mark)
            return;
        Node *afterNode = prevAtref.ptr;
        Node *afterBeforeref = afterNode->before;
        Node *afterBeforeNode = afterBeforeref;
```

```
      if (afterBeforeNode == previous)
        return;
      DPointer<doublylinked::Node, sizeof(size_t)> afterAtref = afterNode->after;
      if (afterAtref == NULL && !afterAtref.mark)
        afterNode->before = previous;
  }
};
```

2.2.3 Hash Set

```
class Lockprogram {
public:
   class LockFreehash {
   public:
      class Node {

      public:
         int item;
         int key; //item's hash code
         DPointer<LockFreehash::Node, sizeof(size_t)> next;
         Node() {
         }
         Node(int item1) { // usual constructor
            this->key = item1;
            this->next = DPointer<LockFreehash::Node, sizeof(size_t)>();
         }
         Node(int key, int x) {
            this->key = key;
            this->item = x;
            this->next = DPointer<LockFreehash::Node, sizeof(size_t)>();
         }

         Node* getnext() {
            bool cMarked[] = { false };
            bool sMarked[] = { false };
            Node* succ;
            Node* entry = this->next.ptr;
            cMarked[0] = this->next.mark;
            while (cMarked[0]) {
               succ = entry->next.ptr;
               sMarked[0] = entry->next.mark;
               this->next.ptr = succ;
               this->next.mark = sMarked[0];
               entry = this->next.ptr;
               cMarked[0] = this->next.mark;
            }
            return entry;
         }
      };
   public:
      class Window {
      public:
         Node *pred;
         Node *curr;
         Window() {
         }
         Window(Node *pred, Node *curr) {
            this->pred = pred;
            this->curr = curr;
         }
      };
      const static int WORD_SIZE = 24;
      const static int LO_MASK = 0x00000001;
      const static int HI_MASK = 0x00800000;
      const static int MASK = 0x00FFFFFF;
```

```
    Node *head;
    LockFreehash() {
       this->head = new Node(0);
       Node *tail = new Node(2147483647);
       while (!head->next.cas(DPointer<LockFreehash::Node,
            sizeof(size_t)>(tail, 0),NULL));
    }
    LockFreehash(Node* e) {
       this->head = e;
    }

    int hashcode(int x) {
       std::tr1::hash<int> hash_fn;
       int a = hash_fn(x);
       return a & MASK;
    }

    int reverse(int key) {
       int loMask = LO_MASK;
       int hiMask = HI_MASK;
       int result = 0;
       for (int i = 0; i < WORD_SIZE; i++) {
          if ((key & loMask) != 0) { // bit set
             result |= hiMask;
          }
          loMask <<= 1;
          hiMask >>= 1; // fill with 0 from left
       }
       return result;
    }

    int makeRegularKey(int x) {
       std::tr1::hash<int> hash_fn;
       int a = hash_fn(x);
       int code = a & MASK; // take 3 lowest bytes
       return reverse(code | HI_MASK);
    }

    int makeSentinelKey(int key) {
       return reverse(key & MASK);
    }

    Window* find(Node* head, int key) {
       Node* pred = head;
       Node* curr = head->getnext();
       while (curr->key < key) {
          pred = curr;
          curr = pred->getnext();
       }
       return new Window(pred, curr);
    }

    bool add(int x) {
       int key = makeRegularKey(x);
       bool splice;
       while (true) {
          Window* window = find(head, key);
```

```
         Node* pred = window->pred;
         Node* curr = window->curr;
         Node* entry = new Node(key, x);
         entry->next = DPointer < LockFreehash::Node, sizeof(size_t) > (curr,
             0);
         splice = pred->next.cas(DPointer<Node, sizeof(size_t)>(entry, 0),
             curr);
         if (splice)
            return true;
         else
            continue;
      }
   }

   bool remove(int x) {
      int key = makeRegularKey(x);
      bool snip;
      while (true) {
         Window* window = find(head, key);
         Node* pred = window->pred;
         Node* curr = window->curr;
         if (curr->key != key) {
            return false;
         } else {
            snip = pred->next.cas(DPointer<Node, sizeof(size_t)>(curr, true),
                curr);
            if (snip)
               return true;
            else
               continue;
         }
      }
   }

   bool contains(int x) {
      int key = makeRegularKey(x);
      Window* window = find(head, key);
      Node* pred = window->pred;
      Node* curr = window->curr;
      return curr->key == key;
   }

   LockFreehash* getsentinel(int index) {
      int key = makeSentinelKey(index);
      bool splice;
      while (true) {
         Window* window = find(head, key);
         Node* pred = window->pred;
         Node* curr = window->curr;
         // is the key present?
         if (curr->key == key) {
            return new LockFreehash(curr);
         } else {
            // splice in new entry
            Node* entry = new Node(key);
            entry->next = DPointer < LockFreehash::Node, sizeof(size_t) >
                (pred, 0);
```

```
            splice = pred->next.cas(DPointer<Node, sizeof(size_t)>(entry,
                false), curr);
            if (splice) {
               return new LockFreehash(curr);
            } else
               continue;
         }
      }
   }
};

vector<LockFreehash*> bucket;
int bucketSize;
int setSize;
const static double THRESHOLD = 4.0;
LockFreehash lfh;
Lockprogram() {
}
Lockprogram(int capacity) {

   for (int i = 0; i < capacity; i++) {
      LockFreehash* temp = new LockFreehash();
      this->bucket.push_back(temp);
   }
   this->bucketSize = 2;
   this->setSize = 0;
}

LockFreehash* getBucketList(int myBucket) {
   if (this->bucket[myBucket] == NULL
      )
      initializeBucket(myBucket);
   return this->bucket[myBucket];
}

int getparent(int myBucket) {
   int parent = this->bucketSize;
   do {
      parent = parent >> 1;
   } while (parent > myBucket);
   parent = myBucket - parent;
   return parent;
}

void initializeBucket(int myBucket) {
   int parent = getparent(myBucket);
   if (this->bucket[parent] = NULL
      )
      initializeBucket(parent);
   LockFreehash* b = this->bucket[parent]->getsentinel(myBucket);
   if (b != NULL
      )
      this->bucket[myBucket] = b;
}

bool add(int x) {
   int mybucket = abs(lfh.hashcode(x) % this->bucketSize);
```

```
    LockFreehash* b = getBucketList(mybucket);
    if (!b->add(x))
      return false;
    int setSizeNow = this->setSize + 1;
    int bucketSizeNow = this->bucketSize;
    if (setSizeNow / (double) bucketSizeNow > THRESHOLD)
      this->bucketSize = 2 * bucketSizeNow;
    return true;
  }

  bool remove(int x) {
    int myBucket = abs(lfh.hashcode(x) % bucketSize);
    LockFreehash* b = getBucketList(myBucket);
    if (!b->remove(x))
      return false;
    return true;
  }

  bool contains(int x) {
    int myBucket = abs(lfh.hashcode(x) % bucketSize);
    LockFreehash* b = getBucketList(myBucket);
    return b->contains(x);

  }
};
```

2.2.4 List

```
class LockFreeList {
public:
   class Node {

   public:
      int item;
      int key; //item's hash code

      DPointer<LockFreeList::Node, sizeof(size_t)> next;
      Node() {
      }
      Node(int item1) { // usual constructor
         this->item = item1;
         this->key = item1; //instead of hashcode(), we have used the item itself
             as the key.
         this->next = DPointer<LockFreeList::Node, sizeof(size_t)>();
      }
   };

   Node *head;

   LockFreeList() {
      this->head = new Node(0);
      Node *tail = new Node(1000000);
      while (!head->next.cas(DPointer<LockFreeList::Node, sizeof(size_t)>(tail,
          0), NULL));
   }

   bool add(int item) {
      int key = item;
      bool splice;
      while (true) {
         Window *window = find(head, key);
         Node *pred = window->pred, *curr = window->curr;
         Node *node = new Node(item);
         node->next = DPointer < LockFreeList::Node, sizeof(size_t) > (curr, 0);
         if (pred->next.cas(DPointer<LockFreeList::Node, sizeof(size_t)>(node,
             0), curr)) {
            return true;
         }
      }
   }

   bool remove(int item) {
      int key = item;
      bool snip;
      while (true) {
         Window *window = find(head, key);
         Node *pred = window->pred, *curr = window->curr;
         Node *succ = curr->next.ptr;
         snip = curr->next.cas(DPointer<LockFreeList::Node, sizeof(size_t)>(succ,
             1),succ);
         if (!snip)
            continue;
```

```
      pred->next.cas(DPointer<LockFreeList::Node, sizeof(size_t)>(succ, 0),
           curr);
      return true;
    }
  }

  bool contains(int item) {
    int key = item;
    Window *window = find(head, key);
    Node *pred = window->pred, *curr = window->curr;
    return (curr->key == key);
  }

  class Window {
  public:
    Node *pred;
    Node *curr;
    Window(Node *pred, Node *curr) {
      this->pred = pred;
      this->curr = curr;
    }
  };

  LockFreeList::Window* find(Node *head, int key) {
    Node *pred = NULL, *curr = NULL, *succ = NULL;
    bool marked[] = { false }; // is curr marked?
    bool snip;
    int flag = 0;
    retry: while (true) {
      pred = head;
      curr = pred->next.ptr;
      while (true) {
        succ = curr->next.ptr;
        marked[0] = curr->next.mark;

        while (marked[0]) { // replace curr if marked
          snip = pred->next.cas(DPointer<Node, sizeof(size_t)>(succ,
               false),curr);
          if (!snip) {
            goto retry;
          }
          curr = pred->next.ptr;
          succ = curr->next.ptr;
          marked[0] = curr->next.mark;
        }
        if (curr->key >= key)
          return new LockFreeList::Window(pred, curr);
        pred = curr;
        curr = succ;
      }
    }
  }
};
```

2.2.5 Queue

```
class Queue {
   class Node {
   public:
      int value;
      Node *next;

      Node(int value) {
         this->value = value;
         this->next = NULL;
      }
   };
private:
   Node *head;
   Node *tail;
public:

   Queue() {
      Node *sentinel = new Node(-1);
      this->head = sentinel;
      this->tail = sentinel;
   }

public:
   void enqueue(int item) {
      Node *node = new Node(item);
      Node *last, *next;

      while (true) {
         last = tail; // read tail
         next = last->next;
         if (last == tail) {
            if (next == NULL)
            {
               if
                    (reinterpret_cast<Node*>(compareAndExchange(reinterpret_cast<volatile
                    size_t*>(&last->next), reinterpret_cast<size_t>(next),
                    reinterpret_cast<size_t>(node))) == next) {
                  compareAndExchange(reinterpret_cast<volatile size_t*>(&tail),
                      reinterpret_cast<size_t>(last),
                      reinterpret_cast<size_t>(node));
                  return;
               }
            } else {
               compareAndExchange(reinterpret_cast<volatile size_t*>(&tail),
                    reinterpret_cast<size_t>(last),
                    reinterpret_cast<size_t>(next));
            }
         }
      }
   }

   int dequeue() {
      int c = 0;
      while (true) {
         Node *first = head;
```

```
        Node *last = tail;
        Node *next = first->next;
        if (first == head) { // are they consistent?
           if (first == last) { // is queue empty or tail falling behind?
              if (next == NULL || head->value == -1) { // is queue empty?
                 cout << "\nqueue is empty";
                 return -1;
              }
              compareAndExchange(reinterpret_cast<volatile size_t*>(&tail),
                   reinterpret_cast<size_t>(last),
                   reinterpret_cast<size_t>(next));
           } else {
              int value = next->value;
              if
                   (reinterpret_cast<Node*>(compareAndExchange(reinterpret_cast<volatile
                   size_t*>(&head), reinterpret_cast<size_t>(first),
                   reinterpret_cast<size_t>(next))) == first)
                 return value;
           }
        }
     }
  }
};
```

2.2.6 Stack

```
struct Node {
   int value;
   struct Node* next;
};

struct Node* top = NULL;

bool tryPush(Node* node) {
   Node* oldTop = top;
   node->next = oldTop;
   return (reinterpret_cast<struct Node*>(compareAndExchange(
        reinterpret_cast<volatile size_t*>(&top),
        reinterpret_cast<size_t>(oldTop), reinterpret_cast<size_t>(node))));
}

void push(int value) {
   Node* node = (struct Node*) malloc(sizeof(struct Node));
   while (true) {
      if (tryPush(node)) {
         return;
      }
   }
}

Node* tryPop() {
   Node* oldTop = top;
   if (oldTop == NULL)
   {
      cout << "\nEmpty Stack";
      return NULL;
   }
   Node* newTop = oldTop->next;
   if (reinterpret_cast<struct
        Node*>(compareAndExchange(reinterpret_cast<volatile size_t*>(&top),
        reinterpret_cast<size_t>(oldTop), reinterpret_cast<size_t>(newTop))) ==
        oldTop) {
      return oldTop;
   } else {
      return NULL;
   }
}

int pop() {
   while (true) {
      Node *returnNode = tryPop();
      if (returnNode != NULL)
      {
         return returnNode->value;
      }
   }
}
```

2.3 MCAS-based Lock-free Implementation

Library Used by all Benchmarks

This library is essentially a software implementation of the MCAS primitive. Calls to the `mcas()` function are replaced with a hardware MCAS call during simulation.

```
#define LOCKTABLESIZE 819412

int lock[LOCKTABLESIZE];
void sort(volatile size_t *list[], volatile size_t oldV[],
      volatile size_t newV[], int length) {
   for (int i = 0; i < length; i++) {
      for (int j = 0; j < length - (i + 1); j++) {
         if (list[j] > list[j + 1]) {
            volatile size_t *temp = list[j + 1];
            list[j + 1] = list[j];
            list[j] = temp;
            volatile size_t tV = oldV[j + 1];
            oldV[j + 1] = oldV[j];
            oldV[j] = tV;
            tV = newV[j + 1];
            newV[j + 1] = newV[j];
            newV[j] = tV;
         }
      }
   }
}

void initialiseLockArray() {
   int i;
   for (i = 0; i < LOCKTABLESIZE; i++) {
      int falseValue = 0;
      lock[i] = falseValue;
   }
}

void acquireLock(volatile size_t *location) {
   volatile size_t index = (volatile size_t) location;

   int hashIndex = index % LOCKTABLESIZE;
   int falseValue = 0;
   int trueValue = 1;
   while ((compareAndExchange(
         reinterpret_cast<volatile size_t*>(&lock[hashIndex]), falseValue,
         trueValue)) != falseValue) {
   }

}

void releaseLock(volatile size_t *location) {
   volatile size_t index = ((volatile size_t) location) % LOCKTABLESIZE;
   int hashIndex = index % LOCKTABLESIZE;
   int falseValue = 0;
   lock[hashIndex] = falseValue;

```

```
}
void fail(volatile size_t **addressArray, unsigned int index) {
  unsigned int i;
  for (i = index; i > 0; i--) {
    releaseLock(addressArray[i]);
  }
}

bool mcas(int n, volatile size_t *m0, volatile size_t old0,
    volatile size_t new0) {
  initialiseLockArray();
  int i;
  volatile size_t *address[1] = { m0 };
  volatile size_t oldV[1] = { old0 };
  volatile size_t newV[1] = { new0 };
  sort(address, oldV, newV, 1);

  for (i = 0; i < n; i++) {
    acquireLock(address[i]);
    if ((unsigned long int) *address[i] != (unsigned long int) oldV[i]) {
      fail(address, i);
      return false;
    }
  }

  if (i == n) {
    for (i = n - 1; i >= 0; i--) {
      *address[i] = newV[i];
      releaseLock(address[i]);
    }
    return true;
  }
}

bool mcas(int n, volatile size_t *m0, volatile size_t *m1, volatile size_t old0,
    volatile size_t old1, volatile size_t new0, volatile size_t new1) {
  initialiseLockArray();
  int i;
  volatile size_t *address[2] = { m0, m1 };
  volatile size_t oldV[2] = { old0, old1 };
  volatile size_t newV[2] = { new0, new1 };
  sort(address, oldV, newV, 2);

  for (i = 0; i < n; i++) {
    acquireLock(address[i]);
    if ((unsigned long int) *address[i] != (unsigned long int) oldV[i]) {
      fail(address, i);
      return false;
    }
  }

  if (i == n) {
    for (i = n - 1; i >= 0; i--) {
      *address[i] = newV[i];
      releaseLock(address[i]);
    }
    return true;
```

```
    }
}

bool mcas(int n, volatile size_t *m0, volatile size_t *m1, volatile size_t *m2,
        volatile size_t old0, volatile size_t old1, volatile size_t old2,
        volatile size_t new0, volatile size_t new1, volatile size_t new2) {
    initialiseLockArray();
    int i;
    volatile size_t *address[3] = { m0, m1, m2 };
    volatile size_t oldV[3] = { old0, old1, old2 };
    volatile size_t newV[3] = { new0, new1, new2 };
    sort(address, oldV, newV, n);

    for (i = 0; i < n; i++) {
        acquireLock(address[i]);
        if ((unsigned long int) *address[i] != (unsigned long int) oldV[i]) {
            fail(address, i);
            return false;
        }
    }

    if (i == n) {
        for (i = n - 1; i >= 0; i--) {
            *address[i] = newV[i];
            releaseLock(address[i]);
        }
        return true;
    }
}

bool mcas(int n, volatile size_t *m0, volatile size_t *m1, volatile size_t *m2,
        volatile size_t *m3, volatile size_t old0, volatile size_t old1,
        volatile size_t old2, volatile size_t old3, volatile size_t new0,
        volatile size_t new1, volatile size_t new2, volatile size_t new3) {
    initialiseLockArray();
    int i;
    volatile size_t *address[4] = { m0, m1, m2, m3 };
    volatile size_t oldV[4] = { old0, old1, old2, old3 };
    volatile size_t newV[4] = { new0, new1, new2, new3 };
    sort(address, oldV, newV, n);

    for (i = 0; i < n; i++) {
        acquireLock(address[i]);
        if ((unsigned long int) *address[i] != (unsigned long int) oldV[i]) {
            fail(address, i);
            return false;
        }

    }

    if (i == n) {
        for (i = n - 1; i >= 0; i--) {
            *address[i] = newV[i];
            releaseLock(address[i]);
        }
        return true;
    }
```

```
}

bool mcas(int n, volatile size_t *m0, volatile size_t *m1, volatile size_t *m2,
      volatile size_t *m3, volatile size_t *m4, volatile size_t *m5,
      volatile size_t old0, volatile size_t old1, volatile size_t old2,
      volatile size_t old3, volatile size_t old4, volatile size_t old5,
      volatile size_t new0, volatile size_t new1, volatile size_t new2,
      volatile size_t new3, volatile size_t new4, volatile size_t new5) {
   initialiseLockArray();
   int i;
   volatile size_t *address[6] = { m0, m1, m2, m3, m4, m5 };
   volatile size_t oldV[6] = { old0, old1, old2, old3, old4, old5 };
   volatile size_t newV[6] = { new0, new1, new2, new3, new4, new5 };
   sort(address, oldV, newV, n);

   for (i = 0; i < n; i++) {
      acquireLock(address[i]);
      if ((unsigned long int) *address[i] != (unsigned long int) oldV[i]) {
         fail(address, i);
         return false;
      }

   }

   if (i == n) {
      for (i = n - 1; i >= 0; i--) {
         *address[i] = newV[i];
         releaseLock(address[i]);
      }
      return true;
   }
}
```

2.3.1 Binary Search Tree

```
struct bst {
  int data;
  struct bst *left;
  struct bst *right;
};
typedef struct bst bst_t;
bst_t *root = NULL;

bst_t *get_new_node(int val) {
  bst_t *node = (bst_t *) malloc(sizeof(bst_t));
  node->data = val;
  node->left = NULL;
  node->right = NULL;
  return node;
}

bst_t *insert(bst_t *root, int val) {
  if (!root)
    return get_new_node(val);
  bst_t *prev = NULL, *ptr = root, *last, *temp, *node;
  node = get_new_node(val);
  char type = ' ';
  while (true) {
    while (ptr) {
      prev = ptr;
      if (val < ptr->data) {
        ptr = ptr->left;
        type = 'l';
      } else {
        ptr = ptr->right;
        type = 'r';
      }
    }
    if (type == 'l') {
      last = prev->left;
      if(mcas(1, reinterpret_cast<volatile size_t*>(&prev->left),
          reinterpret_cast<volatile size_t>(last), reinterpret_cast<volatile
          size_t>(node)))
        return root;
    } else {
      last = prev->right;
      if(mcas(1, reinterpret_cast<volatile size_t*>(&prev->right),
          reinterpret_cast<volatile size_t>(last), reinterpret_cast<volatile
          size_t>(node)))
        return root;
    }
  }
}

int find_minimum_value(bst_t *ptr) {
  int min = ptr ? ptr->data : 0;
  while (ptr) {
    if (ptr->data < min)
      min = ptr->data;
    if (ptr->left) {
```

```
            ptr = ptr->left;
         } else if (ptr->right) {
            ptr = ptr->right;
         } else
            ptr = NULL;
      }
      return min;
   }

   bst_t *deleter(bst_t *root, int val) {
      bst_t *prev = NULL, *ptr = root, *last, *present, *temp;
      char type = ' ';
      while (ptr) {
         if (ptr->data == val) {
            if (!ptr->left && !ptr->right) { // node to be removed has no children's
               if (ptr != root && prev) { // delete leaf node
                  if (type == 'l') {
                     while (flag1) {
                        last = prev;
                        present = ptr;
                        temp = NULL;
                        if(mcas(2, reinterpret_cast<volatile size_t*>(&ptr),
                              reinterpret_cast<volatile size_t*>(&prev->left),
                              reinterpret_cast<volatile size_t>(present),
                              reinterpret_cast<volatile size_t>(present),
                              reinterpret_cast<volatile size_t>(temp),
                              reinterpret_cast<volatile size_t>(temp)))
                           return root;
                     }
                  } else {
                     while (flag1) {
                        last = prev;
                        present = ptr;
                        temp = NULL;
                        if(mcas(2, reinterpret_cast<volatile size_t*>(&ptr),
                              reinterpret_cast<volatile size_t*>(&prev->right),
                              reinterpret_cast<volatile size_t>(present),
                              reinterpret_cast<volatile size_t>(present),
                              reinterpret_cast<volatile size_t>(temp),
                              reinterpret_cast<volatile size_t>(temp)))
                           return root;
                     }
                  }
               } else {
                  last = root;
                  temp = NULL;
                  if(mcas(1, reinterpret_cast<volatile size_t*>(&root),
                        reinterpret_cast<volatile size_t>(last),
                        reinterpret_cast<volatile size_t>(temp)))
                     return root;
               } // deleted node is root
            } else if (ptr->left && ptr->right) { // node to be removed has two
                  children's
               ptr->data = find_minimum_value(ptr->right); // find minimum value
                   from right subtree
               val = ptr->data;
               prev = ptr;
```

```
            ptr = ptr->right; // continue from right subtree delete min node
            type = 'r';
            continue;
         } else { // node to be removed has one children
            if (ptr == root) { // root with one child
               if (root->left) {
                  last = root;
                  temp = root->left;
                  if(mcas(1, reinterpret_cast<volatile size_t*>(&root),
                       reinterpret_cast<volatile size_t>(last),
                       reinterpret_cast<volatile size_t>(temp)))
                     return root;
               } else {
                  last = root;
                  temp = root->right;
                  if(mcas(1, reinterpret_cast<volatile size_t*>(&root),
                       reinterpret_cast<volatile size_t>(last),
                       reinterpret_cast<volatile size_t>(temp)))
                     return root;
               }
            } else { // subtree with one child
               if (type == 'l') {
                  if (ptr->left) {
                     while (flag1) {
                        present = ptr;
                        temp = NULL;
                        if(mcas(2, reinterpret_cast<volatile
                             size_t*>(&prev->left), reinterpret_cast<volatile
                             size_t*>(&ptr), reinterpret_cast<volatile
                             size_t>(present), reinterpret_cast<volatile
                             size_t>(present), reinterpret_cast<volatile
                             size_t>(present->left), reinterpret_cast<volatile
                             size_t>(temp)))
                           return root;
                     }
                  } else {
                     present = ptr;
                     temp = NULL;
                     if(mcas(2, reinterpret_cast<volatile size_t*>(&prev->right),
                          reinterpret_cast<volatile size_t*>(&ptr),
                          reinterpret_cast<volatile size_t>(present),
                          reinterpret_cast<volatile size_t>(present),
                          reinterpret_cast<volatile size_t>(present->right),
                          reinterpret_cast<volatile size_t>(temp)))
                        return root;
                  }
               } else {
                  if (ptr->left) {
                     last = ptr->left;
                     present = ptr;
                     temp = NULL;
                     if(mcas(2, reinterpret_cast<volatile size_t*>(&prev->left),
                          reinterpret_cast<volatile size_t*>(&ptr),
                          reinterpret_cast<volatile size_t>(present),
                          reinterpret_cast<volatile size_t>(present),
                          reinterpret_cast<volatile size_t>(last),
                          reinterpret_cast<volatile size_t>(temp)))
```

```
                    return root;
                } else {
                    last = ptr->right;
                    present = ptr;
                    temp = NULL;
                    if(mcas(2, reinterpret_cast<volatile size_t*>(&prev->right),
                        reinterpret_cast<volatile size_t*>(&ptr),
                        reinterpret_cast<volatile size_t>(present),
                        reinterpret_cast<volatile size_t>(present),
                        reinterpret_cast<volatile size_t>(last),
                        reinterpret_cast<volatile size_t>(temp));
                    return root;
                }
            }
        }
    }
        }
        prev = ptr;
        if (val < ptr->data) {
            ptr = ptr->left;
            type = 'l';
        } else {
            ptr = ptr->right;
            type = 'r';
        }
    }
    return root;
}

void enq(int value) {
    root = insert(root, value);
}

void deq(int value) {
    root = deleter(root, value);
}
```

2.3.2 Doubly Linked List

```
struct node {
   int data;
   struct node *next;
   struct node *prev;
};
struct node* head = NULL;

void insertAfter(struct node* prev_node, int new_data) {
   struct node* new_node = (struct node*) malloc(sizeof(struct node));
   new_node->data = new_data;
   new_node->next = NULL;
   new_node->prev = NULL;
   struct node *temp = NULL, *next_l, *prev_l;
   while (true) {
      next_l = prev_node->next;
      prev_l = prev_node;

      if(mcas(4, reinterpret_cast<volatile size_t*>(&new_node->next),
           reinterpret_cast<volatile size_t*>(&new_node->prev),
           reinterpret_cast<volatile size_t*>(&prev_node->next),
           reinterpret_cast<volatile size_t*>(&prev_node->next->prev),
           reinterpret_cast<volatile size_t>(temp), reinterpret_cast<volatile
           size_t>(temp), reinterpret_cast<volatile size_t>(next_l),
           reinterpret_cast<volatile size_t>(prev_node),
           reinterpret_cast<volatile size_t>(next_l), reinterpret_cast<volatile
           size_t>(prev_node), reinterpret_cast<volatile size_t>(new_node),
           reinterpret_cast<volatile size_t>(new_node)))
         return;
   }
}

void insert(int key) {
   struct node *prev = head, *curr = prev;
   while (curr->data < key) {
      prev = curr;
      curr = curr->next;
   }
   insertAfter(prev, key);
}

void deleteNode(struct node *del) {
   struct node *prev_l, *next_l;
   while (true) {
      prev_l = del->next->prev;
      next_l = del->prev->next;

      if(mcas(2, reinterpret_cast<volatile size_t*>(&del->next->prev),
           reinterpret_cast<volatile size_t*>(&del->prev->next),
           reinterpret_cast<volatile size_t>(prev_l), reinterpret_cast<volatile
           size_t>(next_l), reinterpret_cast<volatile size_t>(del->prev),
           reinterpret_cast<volatile size_t>(del->next)))
         return;
   }
   free(del);
}
```

```
void deleten(int key) {
   struct node *prev = head, *curr = prev;
   while (curr->data < key) {
      prev = curr;
      curr = curr->next;
   }
   deleteNode(curr);
}
```

2.3.3 Hash Set

```
static vector<struct node* > table;
struct node {
   int data;
   struct node *next;
};

void linkadd(int num, int index) {
   bool mcast = false;
   struct node *temp, *a1, *a2;
   temp = (struct node *) malloc(sizeof(struct node));
   temp->data = num;
   while (!mcast) {
      if (table[index] == NULL) {
         table[index] = temp;
         table[index]->next = NULL;
         return;
      } else {
         a1 = temp->next;
         a2 = table[index];
         mcast = mcas(2, reinterpret_cast<volatile size_t*>(&temp->next),
              reinterpret_cast<volatile size_t*>(&(table[index])),
              reinterpret_cast<volatile size_t>(a1),    reinterpret_cast<volatile
              size_t>(a2), reinterpret_cast<volatile size_t>(table[index]),
              reinterpret_cast<volatile size_t>(temp));
      }
   }
}

bool linkdelete(int num, int index) {
   struct node *temp, *prev;
   while (true) {
      temp = table[index];
      if (temp != NULL)
      {
         if (mcas(1, reinterpret_cast<volatile size_t*>(&(table[index])),
              reinterpret_cast<volatile size_t>(temp), reinterpret_cast<volatile
              size_t>(temp->next)))
            return true;
      } else {
         return false;
      }
   }
}

bool linkcontain(int num, int index) {
   if (table[index] == NULL) {
      return false;
   }
   struct node* r = table[index];
   while (r != NULL) {
      if (r->data == num)
         return true;
      r = r->next;
   }
   return false;
```

```
}

class CoarseHashSet {
   int size;
public:
   CoarseHashSet(int capacity) {
      size = 0;
      table.clear();
      for (int i = 0; i < capacity; i++) {
         struct node *temp = NULL;
         //temp=(struct node *)malloc(sizeof(struct node));
         table.push_back(temp);
      }
   }

   bool contains(int x) {
      std::tr1::hash<int> hash_fn;
      int a = hash_fn(x);
      bool ans = linkcontain(x, abs(a % table.size()));
      return ans;
   }

   void add(int x) {
      std::tr1::hash<int> hash_fn;
      int a = hash_fn(x);
      linkadd(x, abs(a % table.size()));
      size = size + 1;
   }

   void remove(int x) {
      std::tr1::hash<int> hash_fn;
      int a = hash_fn(x);
      bool result = linkdelete(x, abs(a % table.size()));
      size = result ? size - 1 : size;
   }
};
```

2.3.4 List

```
class List {
public:
   class Node {
   public:
      int item;
      int key;
      Node *next;
      Node(int item) {
         this->item = item;
         this->key = item;
      }
   };
   Node *head;
   Node *tail;

   List() {
      head = new Node(-2147483648); // Add sentinels to start and end
      tail = new Node(2147483647);
      head->next = this->tail;
   }

   bool add(int item) {
      Node *pred, *curr, *last, *temp;
      int key = item;
      bool flag, mcast = false;
      Node *node = new Node(item);
      while (true) {
         pred = head;
         curr = pred->next;
         while (curr->key < key) {
            pred = curr;
            curr = curr->next;
         }
         last = curr;
         temp = NULL;
         mcast = mcas(2, reinterpret_cast<volatile size_t*>(&pred->next),
             reinterpret_cast<volatile size_t*>(&node->next),
             reinterpret_cast<volatile size_t>(last), reinterpret_cast<volatile
             size_t>(temp), reinterpret_cast<volatile size_t>(node),
             reinterpret_cast<volatile size_t>(last));
         if (mcast) {
            flag = true;
            return flag;
         }
      }
   }

   bool remove(int item) {
      Node *pred, *curr, *last, *temp, *prev;
      int key = item;
      while (true) {
         pred = this->head;
         curr = pred->next;
         while (curr->key < key) {
            pred = curr;
```

```
            curr = curr->next;
         }
         last = curr->next;
         prev = pred->next;
         mcas(1, reinterpret_cast<volatile size_t*>(&pred->next),
              reinterpret_cast<volatile size_t>(prev), reinterpret_cast<volatile
              size_t>(last));
         return true;
      }
   }

   bool contains(int item) {
      Node *pred, *curr, *last, *prev, *temp;
      int key = item;
      pred = head;
      curr = pred->next;
      while (curr->key < key) {
         prev = pred;
         last = curr;
         mcas(2, reinterpret_cast<volatile size_t*>(&pred),
              reinterpret_cast<volatile size_t*>(&curr),
              reinterpret_cast<volatile size_t>(prev), reinterpret_cast<volatile
              size_t>(last), reinterpret_cast<volatile size_t>(last),
              reinterpret_cast<volatile size_t>(last->next));
      }
      return (key == curr->key);
   }

   int size() {
      Node *pred = head;
      int l = 0;
      while (pred != tail) {
         l++;
         pred = pred->next;
      }
      return l;
   }
};
```

2.3.5 Queue

```
struct node
{
   int data;
   node *next;
}*front = NULL, *rear = NULL, *p = NULL, *np = NULL, *last = NULL, *first =
    NULL, *next1 = NULL;

void enqueue(int x) {
   bool result = false;
   np = new node;
   np->data = x;
   np->next = NULL;
   while (!result) {
      last = rear;
      first = front;
      if (front == NULL) {
         front = rear = np;
      } else {
         next1 = last->next;
         result = mcas(2, reinterpret_cast<volatile size_t*>(&rear->next),
             reinterpret_cast<volatile size_t*>(&rear),
             reinterpret_cast<volatile size_t>(next1), reinterpret_cast<volatile
             size_t>(last), reinterpret_cast<volatile size_t>(np),
             reinterpret_cast<volatile size_t>(np));
      }
   }
}

void dequeue() {
   bool result = false;
   while (true) {
      if (front == NULL) {
         return;
      } else {
         p = front;
         result = mcas(1, reinterpret_cast<volatile size_t*>(&front),
             reinterpret_cast<volatile size_t>(p), reinterpret_cast<volatile
             size_t>(front->next));
         if (result)
            return;
      }
   }
}
```

2.3.6 Stack

We chose to employ an MCAS of arity 2, although a single address CAS is sufficient. We chose to do this simply to exercise our design of the MCAS hardware.

```
class stack {
public:
   class Node {
   public:
      int item;
      Node *next;
      Node(int item) {
         this->item = item;
         this->next = NULL;
      }
   };
   Node *top;

   void push(int value) {
      bool flag = false;
      Node *node = new Node(value);
      Node *topd, *temp = NULL;
      while (true) {
         topd = top;
         flag = mcas(2, reinterpret_cast<volatile size_t*>(&node->next),
              reinterpret_cast<volatile size_t*>(&top), reinterpret_cast<volatile
              size_t>(temp), reinterpret_cast<volatile size_t>(topd),
              reinterpret_cast<volatile size_t>(topd), reinterpret_cast<volatile
              size_t>(node));
         if (flag) {
            return;
         }
      }
   }

   void pop() {
      int t;
      Node *topd;
      int temp;
      bool result = false;
      while (true) {
         if (top != NULL) {
            t = top->item;
            topd = top;
            result = mcas(1, reinterpret_cast<volatile size_t*>(&top),
                 reinterpret_cast<volatile size_t>(topd),
                 reinterpret_cast<volatile size_t>(top->next));
            if (result) {
               return;
            }
         } else {
            return;
         }
      }
   }
};
```

References

[1] P. Martin, "Practical lock-free doubly-linked list," uS Patent 7,533,138. [Online]. Available: http://www.google.co.in/patents/US7533138

[2] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.