

# 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:

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

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**  $B \in N$  **do**  
    | initialize vector  $in_B$  to -1  
**end**  
**repeat**  
    |  $B$  = first node of *worklist*  
    | *worklist* = *worklist* -  $B$   
    | *totaleffect*: initialize vector to -1  
    | **for each**  $P \in Predecessors(B)$  **do**  
    |     | *effect* =  $F(in_B)$   
    |     | *totaleffect* = *totaleffect*  $\sqcap$  *effect*  
    | **end**  
    | **if**  $in_B \neq totaleffect$  **then**  
    |     |  $in_B = totaleffect$   
    |     | *worklist* = *worklist*  $\cup$  *successors*( $B$ )  
    | **end**  
**until** *worklist* =  $\phi$   
*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* MTSs 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.

	<b>Lock-based</b>	<b>CAS-based Lock-free</b>	<b>MCAS-based Lock-free</b>
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.

---

```
1 pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
2 struct bst {
3     int data;
4     struct bst *left;
5     struct bst *right;
6 };
7 typedef struct bst bst_t;
8
9 bst_t *get_new_node(int val) {
10     bst_t *node = (bst_t *) malloc(sizeof(bst_t));
11     node->data = val;
12     node->left = NULL;
13     node->right = NULL;
14     return node;
15 }
16 bst_t *root = NULL;
17
18 bst_t *insert(bst_t *root, int val) {
19     if (!root)
20         return get_new_node(val);
21     bst_t *prev = NULL, *ptr = root;
22     char type = ' ';
23     while (ptr) {
24         prev = ptr;
25         if (val < ptr->data) {
26             ptr = ptr->left;
27             type = 'l';
28         } else {
29             ptr = ptr->right;
30             type = 'r';
31         }
32     }
33     if (type == 'l')
34         prev->left = get_new_node(val);
35     else
36         prev->right = get_new_node(val);
37     return root;
38 }
39
40 int find_minimum_value(bst_t *ptr) {
41     int min = ptr ? ptr->data : 0;
42     while (ptr) {
43         if (ptr->data < min)
44             min = ptr->data;
45         if (ptr->left) {
```

```

46         ptr = ptr->left;
47     } else if (ptr->right) {
48         ptr = ptr->right;
49     } else
50         ptr = NULL;
51     }
52     return min;
53 }
54
55 bst_t *deleter(bst_t *root, int val) {
56     bst_t *prev = NULL, *ptr = root;
57     char type = ' ';
58     while (ptr) {
59         if (ptr->data == val) {
60
61             if (!ptr->left && !ptr->right) { // node to be removed has no children's
62                 if (ptr != root && prev) { // delete leaf node
63                     if (type == 'l')
64                         prev->left = NULL;
65                     else
66                         prev->right = NULL;
67                 } else
68                     root = NULL; // deleted node is root
69             } else if (ptr->left && ptr->right) { // node to be removed has two
69                 children's
70                 ptr->data = find_minimum_value(ptr->right); // find minimum value
71                 from right subtree
72                 val = ptr->data;
73                 prev = ptr;
74                 ptr = ptr->right; // continue from right subtree delete min node
75                 type = 'r';
76                 continue;
77             } else { // node to be removed has one children
78                 if (ptr == root) { // root with one child
79                     root = root->left ? root->left : root->right;
80                 } else { // subtree with one child
81                     if (type == 'l')
82                         prev->left = ptr->left ? ptr->left : ptr->right;
83                     else
84                         prev->right = ptr->left ? ptr->left : ptr->right;
85                 }
86             }
87         }
88         prev = ptr;
89         if (val < ptr->data) {
90             ptr = ptr->left;
91             type = 'l';
92             height++;
93         } else {
94             ptr = ptr->right;
95             type = 'r';
96             height++;
97         }
98     }
99     return root;
100 }

```

```
101 void enq(int value) {
102     pthread_mutex_lock(&mutex1);
103     root = insert(root, value);
104     pthread_mutex_unlock(&mutex1);
105 }
106
107 void deq(int value) {
108     pthread_mutex_lock(&mutex1);
109     root = deleter(root, value);
110     pthread_mutex_unlock(&mutex1);
111 }
```

---

## 2.1.2 Doubly Linked List

---

```
1 pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
2 struct node {
3     int data;
4     struct node *next;
5     struct node *prev;
6 };
7 struct node* head = NULL;
8
9 void insertAfter(struct node* prev_node, int new_data) {
10     if (prev_node == NULL)
11     {
12         printf("the given previous node cannot be NULL");
13         return;
14     }
15     struct node* new_node = (struct node*) malloc(sizeof(struct node));
16     new_node->data = new_data;
17     new_node->next = prev_node->next;
18     prev_node->next = new_node;
19     new_node->prev = prev_node;
20     if (new_node->next != NULL)
21         new_node->next->prev = new_node;
22 }
23
24 void insert(int key) {
25     pthread_mutex_lock(&mutex1);
26     struct node *prev = head, *curr = prev;
27     while (curr->data < key) {
28         prev = curr;
29         curr = curr->next;
30     }
31     insertAfter(prev, key);
32     pthread_mutex_unlock(&mutex1);
33 }
34
35 void deleteNode(struct node *del) {
36     if (del == NULL)
37         return;
38     del->next->prev = del->prev;
39     del->prev->next = del->next;
40     del = NULL;
41     free(del);
42 }
43
44 void deleten(int key) {
45     pthread_mutex_lock(&mutex1);
46     struct node *prev = head, *curr = prev;
47     while (curr->data < key) {
48         prev = curr;
49         curr = curr->next;
50     }
51     deleteNode(curr);
52     pthread_mutex_unlock(&mutex1);
53 }
```

---



### 2.1.3 Hash Set

---

```
1 pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
2 static vector<struct node*> table;
3 struct node {
4     int data;
5     struct node *next;
6 };
7
8 void linkadd(int num, int index) {
9     struct node *temp;
10    temp = (struct node *) malloc(sizeof(struct node));
11    temp->data = num;
12    if (table[index] == NULL)
13    {
14        table[index] = temp;
15        table[index]->next = NULL;
16    } else {
17        temp->next = table[index];
18        table[index] = temp;
19    }
20 }
21
22 bool linkdelete(int num, int index) {
23     struct node *temp, *prev;
24     temp = table[index];
25     if (temp != NULL)
26     {
27         table[index] = temp->next;
28         return true;
29     }
30     return false;
31 }
32
33 bool linkcontain(int num, int index) {
34     if (table[index] == NULL)
35     {
36         return false;
37     }
38     struct node* r = table[index];
39     while (r != NULL) {
40         if (r->data == num)
41             return true;
42         r = r->next;
43     }
44     return false;
45 }
46
47 class CoarseHashSet {
48     int size;
49 public:
50     CoarseHashSet(int capacity) {
51         size = 0;
52         for (int i = 0; i < capacity; i++) {
53             struct node *temp = NULL;
54             table.push_back(temp);
55         }
```

```

56     }
57
58     bool contains(int x) {
59         std::tr1::hash<int> hash_fn;
60         int a = hash_fn(x);
61         int myBucket = abs(a % table.size());
62         bool ans = linkcontain(x, myBucket);
63         return ans;
64     }
65
66     void add(int x) {
67         pthread_mutex_lock(&mutex1);
68         std::tr1::hash<int> hash_fn;
69         int a = hash_fn(x);
70         int myBucket = abs(a % table.size());
71         linkadd(x, myBucket);
72         size = size + 1;
73         pthread_mutex_unlock(&mutex1);
74         if (size / table.size() > 4)
75             resize();
76     }
77
78     void remove(int x) {
79         pthread_mutex_lock(&mutex1);
80         std::tr1::hash<int> hash_fn;
81         int a = hash_fn(x);
82         int myBucket = abs(a % table.size());
83         bool result = linkdelete(x, myBucket);
84         size = result ? size - 1 : size;
85         pthread_mutex_unlock(&mutex1);
86     }
87 };

```

---

### 2.1.4 List

---

```
1 pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
2
3 class LockedList {
4
5 public:
6     class Node {
7     public:
8         int item;
9         int key;
10        Node *next;
11        Node(int item) {
12            this->item = item;
13            this->key = item;
14        }
15    };
16    Node *head;
17    Node *tail;
18
19    LockedList() {
20        // Add sentinels to start and end
21        head = new Node(INT_MIN);
22        tail = new Node(INT_MAX);
23        head->next = this->tail;
24    }
25
26    bool add(int item) {
27        Node *pred, *curr;
28        int key = item;
29        bool flag;
30
31        pthread_mutex_lock(&mutex1);
32        pred = head;
33        curr = pred->next;
34        while (curr->key < key) {
35            pred = curr;
36            curr = curr->next;
37        }
38
39        Node *node = new Node(item);
40        node->next = curr;
41        pred->next = node;
42        flag = true;
43        pthread_mutex_unlock(&mutex1);
44        return flag;
45    }
46
47    bool remove(int item) {
48        Node *pred, *curr;
49        int key = item;
50        bool flag;
51        pthread_mutex_lock(&mutex1);
52        pred = this->head;
53        curr = pred->next;
54        while (curr->key < key) {
55            pred = curr;
```

```

56         curr = curr->next;
57     }
58     pred->next = curr->next;
59     flag = true;
60     pthread_mutex_unlock(&mutex1);
61     return flag;
62 }
63
64 bool contains(int item) {
65     Node *pred, *curr;
66     int key = item;
67     pthread_mutex_lock(&mutex1);
68     pred = head;
69     curr = pred->next;
70     while (curr->key < key) {
71         pred = curr;
72         curr = curr->next;
73     }
74     pthread_mutex_unlock(&mutex1);
75     return (key == curr->key);
76 }
77 };

```

---

### 2.1.5 Queue

---

```
1 pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
2 struct qNode {
3     int value;
4     struct qNode *next;
5 };
6
7 struct qNode *front = NULL, *rear = NULL;
8
9 void enq(int item) {
10     pthread_mutex_lock(&mutex1);
11     struct qNode *newN = new struct qNode;
12     newN->value = item;
13     newN->next = NULL;
14     if (front == NULL && rear == NULL) {
15         front = newN;
16         rear = newN;
17     } else {
18         rear->next = newN;
19         rear = newN;
20     }
21     pthread_mutex_unlock(&mutex1);
22 }
23
24 int deq() {
25     int t;
26     pthread_mutex_lock(&mutex1);
27     if (front != NULL) {
28         struct qNode *temp = front;
29         t = temp->value;
30         if (front == rear) {
31             front = NULL;
32             rear = NULL;
33         } else
34             front = front->next;
35     } else {
36         cout << "queue is empty" << endl;
37     }
38     pthread_mutex_unlock(&mutex1);
39     return t;
40 }
```

---

### 2.1.6 Stack

---

```
1 pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
2 class stack {
3 public:
4     class Node {
5     public:
6         int item;
7         Node *next;
8         Node(int item) {
9             this->item = item;
10            this->next = NULL;
11        }
12    };
13    Node *top;
14
15    void push(int value) {
16        Node *node = new Node(value);
17        pthread_mutex_lock(&mutex1);
18        node->next = top;
19        top = node;
20        pthread_mutex_unlock(&mutex1);
21    }
22
23
24    void pop() {
25        int t;
26        pthread_mutex_lock(&mutex1);
27        if (top != NULL) {
28            t = top->item;
29            top = top->next;
30        }
31        pthread_mutex_unlock(&mutex1);
32    }
33 };
```

---

## 2.2 CAS-based Lock-free Implementation

### Library Used by all Benchmarks

---

```
1 static size_t compareAndExchange( volatile size_t* addr, size_t oldval, size_t
   newval ){
2     size_t ret;
3     __asm__ volatile( "lock cmpxchg %2, %1\n\t": "=a"(ret), "+m"(*addr):
        "r"(newval), "0"(oldval): "memory" );
4     return ret;
5 }
6
7 static size_t compareAndExchangeTid( volatile size_t* addr, int oldval, int
   newval ){
8     int ret;
9     __asm__ volatile( "lock cmpxchg %2, %1\n\t": "=a"(ret), "+m"(*addr):
        "r"(newval), "0"(oldval): "memory" );
10    return ret;
11 }
12 static size_t AtomicExchange(volatile size_t* ptr, size_t new_value) {
13     __asm__ __volatile__( "xchg %1,%0": "=r" (new_value): "m" (*ptr), "0"
        (new_value): "memory");
14     return new_value; // Now it's the previous value.
15 }
16
17 template<typename T,unsigned N=sizeof (uint32_t)>
18 struct DPointer {
19 public:
20     union {
21         uint64_t ui;
22         struct {
23             T* ptr;
24             size_t mark;
25         };
26     };
27
28     DPointer() : ptr(NULL), mark(0) {}
29     DPointer(T* p) : ptr(p), mark(0) {}
30     DPointer(T* p, size_t c) : ptr(p), mark(c) {}
31
32     bool cas(DPointer<T,N> const& nval, DPointer<T,N> const & cmp)
33     {
34         bool result;
35         __asm__ __volatile__(
36             "lock cmpxchg8b %1\n\t"
37             "setz %0\n"
38             : "=q" (result)
39             , "+m" (ui)
40             : "a" (cmp.ptr), "d" (cmp.mark)
41             , "b" (nval.ptr), "c" (nval.mark)
42             : "cc"
43         );
44         return result;
45     }
46
47     // We need == to work properly
48     bool operator==(DPointer<T,N> const&x) { return x.ui == ui; }
```

```

49 };
50 };
51
52 template<typename T>
53 struct DPointer <T, sizeof (uint64_t)> {
54 public:
55     union {
56         uint64_t ui[2];
57         struct {
58             T* ptr;
59             size_t mark;
60         } __attribute__((__aligned__( 16 )));
61     };
62
63     DPointer() : ptr(NULL), mark(0) {}
64     DPointer(T* p) : ptr(p), mark(0) {}
65     DPointer(T* p, size_t c) : ptr(p), mark(c) {}
66
67     bool cas(DPointer<T,8> const& nval, DPointer<T,8> const& cmp)
68     {
69         bool result;
70         __asm__ __volatile__ (
71             "lock cmpxchg16b %1\n\t"
72             "setz %0\n"
73             : "=q" ( result )
74             , "+m" ( ui )
75             : "a" ( cmp.ptr ), "d" ( cmp.mark )
76             , "b" ( nval.ptr ), "c" ( nval.mark )
77             : "cc"
78         );
79         return result;
80     }
81
82     // We need == to work properly
83     bool operator==(DPointer<T,8> const&x) { return x.ptr == ptr && x.mark ==
84         mark; }
85 };

```

---



### 2.2.1 Binary Search Tree

---

```
1  class Node {
2  public:
3
4      long key;
5      long value;
6      DPointer<Node, sizeof(size_t)> lChild;
7      DPointer<Node, sizeof(size_t)> rChild;
8      Node() {
9      }
10
11     Node(long key, long value) {
12         this->key = key;
13         this->value = value;
14     }
15
16     Node(long key, long value, DPointer<Node, sizeof(size_t)> lChild
17         , DPointer<Node, sizeof(size_t)> rChild) {
18         this->key = key;
19         this->value = value;
20         this->lChild = lChild;
21         this->rChild = rChild;
22     }
23 };
24
25 class SeekRecord {
26 public:
27     Node *ancestor;
28     Node *successor;
29     Node *parent;
30     Node *leaf;
31
32     SeekRecord() {
33     }
34     SeekRecord(Node *ancestor, Node *successor, Node *parent, Node *leaf) {
35         this->ancestor = ancestor;
36         this->successor = successor;
37         this->parent = parent;
38         this->leaf = leaf;
39     }
40 };
41
42 class LockFreeBST {
43 public:
44     static Node *grandParentHead;
45     static Node *parentHead;
46     static LockFreeBST *obj;
47
48     LockFreeBST() {
49         createHeadNodes();
50     }
51
52     long lookup(long target) {
53         Node *node = grandParentHead;
54         while (node->lChild.ptr != NULL) //loop until a leaf or dummy node is
55             reached
```

```

55     {
56         if (target < node->key) {
57             node = node->lChild.ptr;
58         } else {
59             node = node->rChild.ptr;
60         }
61     }
62
63     if (target == node->key)
64         return (1);
65     else
66         return (0);
67 }
68
69 void add(long insertKey) {
70     int nthChild;
71     Node *node;
72     Node *pnode;
73     SeekRecord *s;
74     while (true) {
75         nthChild = -1;
76         pnode = parentHead;
77         node = parentHead->lChild.ptr;
78         while (node->lChild.ptr != NULL) //loop until a leaf or dummy node is
79             reached
80         {
81             if (insertKey < node->key) {
82                 pnode = node;
83                 node = node->lChild.ptr;
84             } else {
85                 pnode = node;
86                 node = node->rChild.ptr;
87             }
88         }
89         Node *oldChild = node;
90
91         if (insertKey < pnode->key) {
92             nthChild = 0;
93         } else {
94             nthChild = 1;
95         }
96
97         //leaf node is reached
98         if (node->key == insertKey) {
99             //key is already present in tree. So return
100             return;
101         }
102
103         Node *internalNode, *lLeafNode, *rLeafNode;
104         if (node->key < insertKey) {
105             rLeafNode = new Node(insertKey, insertKey);
106             internalNode = new Node(insertKey, insertKey, DPointer<Node,
107                                     sizeof(size_t)>(node, 0), DPointer<Node,
108                                     sizeof(size_t)>(rLeafNode, 0));
109         } else {
110             lLeafNode = new Node(insertKey, insertKey);

```

```

109         internalNode = new Node(node->key, node->key, DPointer<Node,
110             sizeof(size_t)>(lLeafNode, 0), DPointer<Node,
111             sizeof(size_t)>(node, 0));
112     }
113     if (nthChild == 0) {
114         if (pnode->lChild.cas(DPointer<Node, sizeof(size_t)>(internalNode,
115             0), oldChild)) {
116             return;
117         } else {
118             //insert failed; help the conflicting delete operation
119             if (node == pnode->lChild.ptr) { // address has not changed. So
120                 CAS would have failed coz of flag/mark only
121                 //help other thread with cleanup
122                 s = seek(insertKey);
123                 cleanUp(insertKey, s);
124             }
125         }
126     } else {
127         if (pnode->rChild.cas(DPointer<Node, sizeof(size_t)>(internalNode,
128             0), DPointer<Node, sizeof(size_t)>(oldChild, 0))) {
129             return;
130         } else {
131             if (node == pnode->rChild.ptr) {
132                 s = seek(insertKey);
133                 cleanUp(insertKey, s);
134             }
135         }
136     }
137 }
138
139 void remove(long deleteKey) {
140     bool isCleanUp = false;
141     SeekRecord *s;
142     Node *parent;
143     Node *leaf = NULL;
144     while (true) {
145         s = seek(deleteKey);
146         if (!isCleanUp) {
147             leaf = s->leaf;
148             if (leaf->key != deleteKey) {
149                 return;
150             } else {
151                 parent = s->parent;
152                 if (deleteKey < parent->key) {
153                     if (parent->lChild.cas(DPointer<Node, sizeof(size_t)>(leaf, 2),
154                         leaf)) {
155                         isCleanUp = true;
156                         //do cleanup
157                         if (cleanUp(deleteKey, s)) {
158                             return;
159                         }
160                     } else {
161                         if (leaf == parent->lChild.ptr) {
162                             cleanUp(deleteKey, s);
163                         }
164                     }
165                 }
166             }
167         }
168     }
169 }

```

```

160         }
161     } else {
162         if (parent->rChild.cas(DPointer<Node, sizeof(size_t)>(leaf, 2),
163             leaf)) {
164             isCleanUp = true;
165             //do cleanup
166             if (cleanUp(deleteKey, s)) {
167                 return;
168             }
169         } else {
170             if (leaf == parent->rChild.ptr) {
171                 //help other thread with cleanup
172                 cleanUp(deleteKey, s);
173             }
174         }
175     }
176 } else {
177     if (s->leaf == leaf) {
178         //do cleanup
179         if (cleanUp(deleteKey, s)) {
180             return;
181         }
182     } else {
183         //someone helped with my cleanup. So I'm done
184         return;
185     }
186 }
187 }
188 }
189
190 int setTag(int stamp) {
191     switch (stamp) // set only tag
192     {
193     case 0:
194         stamp = 1; // 00 to 01
195         break;
196     case 2:
197         stamp = 3; // 10 to 11
198         break;
199     }
200     return stamp;
201 }
202
203 int copyFlag(int stamp) {
204     switch (stamp) //copy only the flag
205     {
206     case 1:
207         stamp = 0; // 01 to 00
208         break;
209     case 3:
210         stamp = 2; // 11 to 10
211         break;
212     }
213     return stamp;
214 }
215

```



```

263     }
264 }
265
266 if (key < ancestor->key) {
267     siblingStamp = copyFlag(siblingStamp); //copy only the flag
268     oldSuccessor = ancestor->lChild.ptr;
269     oldStamp = ancestor->lChild.mark;
270     return (ancestor->lChild.cas(DPointer<Node, sizeof(size_t)>(sibling,
        siblingStamp), DPointer<Node, sizeof(size_t)>(oldSuccessor,
        oldStamp)));
271 } else {
272     siblingStamp = copyFlag(siblingStamp); //copy only the flag
273     oldSuccessor = ancestor->rChild.ptr;
274     oldStamp = ancestor->rChild.mark;
275     return (ancestor->rChild.cas(DPointer<Node, sizeof(size_t)>(sibling,
        siblingStamp), DPointer<Node, sizeof(size_t)>(oldSuccessor,
        oldStamp)));
276 }
277 }
278
279 SeekRecord* seek(long key) {
280     DPointer<Node, sizeof(size_t)> parentField;
281     DPointer<Node, sizeof(size_t)> currentField;
282     Node *current;
283
284     //initialize the seek record
285     SeekRecord *s = new SeekRecord(grandParentHead, parentHead, parentHead,
        parentHead->lChild.ptr);
286
287     parentField = s->ancestor->lChild;
288     currentField = s->successor->lChild;
289
290     while (currentField.ptr != NULL) {
291         current = currentField.ptr;
292         //move down the tree
293         //check if the edge from the current parent node in the access path is
            tagged
294         if (parentField.mark == 0 || parentField.mark == 2) { // 00, 10 untagged
295             s->ancestor = s->parent;
296             s->successor = s->leaf;
297         }
298         //advance parent and leaf pointers
299         s->parent = s->leaf;
300         s->leaf = current;
301         parentField = currentField;
302         if (key < current->key) {
303             currentField = current->lChild;
304         } else {
305             currentField = current->rChild;
306         }
307     }
308     return s;
309 }
310
311 void createHeadNodes() {
312     long key = LONG_MAX;
313     long value = LONG_MIN;

```

```
314     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));
315     grandParentHead = new Node(key, value, DPointer<Node,
        sizeof(size_t)>(parentHead, 0), DPointer<Node, sizeof(size_t)>(new
        Node(key, value), 0));
316 }
317 };
318 Node * LockFreeBST::grandParentHead = NULL;
319 Node * LockFreeBST::parentHead = NULL;
320 LockFreeBST *LockFreeBST::obj = NULL;
```

---

## 2.2.2 Doubly Linked List

---

```
1 class doublylinked {
2 public:
3     class Node {
4     public:
5         int value;
6         DPointer<doublylinked::Node, sizeof(size_t)> after;
7         Node *before;
8
9         Node() {
10        }
11        Node(int key) {
12            this->value = key;
13            this->before = NULL;
14            this->after = DPointer<doublylinked::Node, sizeof(size_t)>();
15        }
16    };
17    Node *headdummy, *taildummy;
18
19    bool deleten(int key) {
20        Node *pred = headdummy, *curr;
21        curr = pred->after.ptr;
22        while (curr->value < key) {
23            pred = curr;
24            curr = curr->after.ptr;
25        }
26        return deleteNode(curr, true);
27    }
28
29    bool deleteNode(Node *thisNode, bool retry) {
30        DPointer<doublylinked::Node, sizeof(size_t)> nextref;
31        while (true) {
32            nextref = thisNode->after;
33            if (nextref.mark)
34                return false;
35            Node *next = nextref.ptr;
36            if (thisNode->after.cas(DPointer<Node, sizeof(size_t)>(next, true),
37                                next));
38            break;
39            if (!retry)
40                return false;
41        }
42        getBack(thisNode);
43        return true;
44    }
45
46    Node* getBack(Node *refNode) {
47        Node *prevref = refNode->before;
48        Node *currentNode = refNode;
49        while (true) {
50            prevref = currentNode->before;
51            Node *backnode = prevref;
52            DPointer<doublylinked::Node, sizeof(size_t)> backAtref = backnode->after;
53            Node *backAftNode = backAtref.ptr;
54            if (backAtref.mark)
55                currentNode = backnode;
```



```

55         else if (backAftNode == refNode)
56             return backnode;
57         else {
58             Node *maybeback = fixforwarduntil(backnode, refNode);
59             if ((maybeback == NULL) && (backnode->after.mark))
60                 currentNode = backnode;
61             else
62                 return maybeback;
63         }
64     }
65 }
66
67 Node* fixforwarduntil(Node *thisNode, Node *laterNode) {
68     Node *nextnode, *worknode = thisNode;
69     DPointer<doublylinked::Node, sizeof(size_t)> thisnodeAtref, workNodeAtref,
        laternodeAtref;
70
71     while (true) {
72         thisnodeAtref = thisNode->after;
73         if (thisnodeAtref.mark)
74             return NULL;
75         laternodeAtref = laterNode->after;
76         if ((laternodeAtref.ptr != NULL) && (laternodeAtref.mark))
77             return NULL;
78         workNodeAtref = worknode->after;
79         if (workNodeAtref.ptr == NULL)
80             return NULL;
81         if (!(workNodeAtref.mark)) {
82             fixforward(worknode);
83             workNodeAtref = worknode->after;
84         }
85         nextnode = workNodeAtref.ptr;
86         if (nextnode == laterNode)
87             return worknode;
88         else if (nextnode->after == NULL
89             ) return NULL;
90         else
91             worknode = nextnode;
92     }
93 }
94
95 void initialise() {
96     Node *a = new Node(0);
97     headdummy = a;
98     tailedummy = a;
99     Node *b = new Node(1000000000);
100     b->before = a;
101     a->after = DPointer < doublylinked::Node, sizeof(size_t) > (b, 0);
102 }
103
104 bool add(int key) {
105     Node *mynode = new Node(key);
106     Node *pred = headdummy, *curr;
107     curr = pred->after.ptr;
108     while (curr->value < key) {
109         if (pred == curr)
110             break;

```

```

111         pred = curr;
112         curr = curr->after.ptr;
113     }
114     if (headdummy == pred)
115         return insertafter(headdummy, mynode);
116     else
117         insertafter(pred, mynode);
118 }
119
120 bool insertafter(Node *previous, Node *mynode) {
121     while (true) {
122         DPointer<doublylinked::Node, sizeof(size_t)> prevAtref = previous->after;
123         if (prevAtref.mark)
124             return false;
125         Node *prevafter = fixforward(previous);
126         if (insertBetween(mynode, previous, prevafter))
127             return true;
128     }
129 }
130
131 bool insertBetween(Node *thisNode, Node *prev, Node *after) {
132     thisNode->before = prev;
133     thisNode->after = DPointer < doublylinked::Node, sizeof(size_t) > (after,
134         0);
135     if (prev->after.cas(DPointer<Node, sizeof(size_t)>(thisNode, false),
136         after)) {
137         reflectforward(thisNode);
138         return true;
139     }
140     return false;
141 }
142
143 Node* fixforward(Node *thisNode) {
144     DPointer<doublylinked::Node, sizeof(size_t)> thisAtref = thisNode->after;
145     Node *laterNode = thisAtref.ptr;
146     Node *laterLater;
147     while (true) {
148         DPointer<doublylinked::Node, sizeof(size_t)> nextref = laterNode->after;
149         if (nextref == NULL || !nextref.mark) {
150             reflectforward(thisNode);
151             return laterNode;
152         } else {
153             laterLater = nextref.ptr;
154             thisNode->after.cas(DPointer<Node, sizeof(size_t)>(laterLater,
155                 false), laterNode);
156             laterNode = laterLater;
157         }
158     }
159 }
160
161 void reflectforward(Node *previous) {
162     DPointer<doublylinked::Node, sizeof(size_t)> prevAtref = previous->after;
163     if (prevAtref.mark)
164         return;
165     Node *afterNode = prevAtref.ptr;
166     Node *afterBeforeref = afterNode->before;
167     Node *afterBeforeNode = afterBeforeref;

```

```
165         if (afterBeforeNode == previous)
166             return;
167         DPointer<doublylinked::Node, sizeof(size_t)> afterAtref = afterNode->after;
168         if (afterAtref == NULL && !afterAtref.mark)
169             afterNode->before = previous;
170     }
171 };
```

---

### 2.2.3 Hash Set

---

```
1 class Lockprogram {
2 public:
3     class LockFreehash {
4     public:
5         class Node {
6
7         public:
8             int item;
9             int key; //item's hash code
10            DPointer<LockFreehash::Node, sizeof(size_t)> next;
11            Node() {
12            }
13            Node(int item1) { // usual constructor
14                this->key = item1;
15                this->next = DPointer<LockFreehash::Node, sizeof(size_t)>();
16            }
17            Node(int key, int x) {
18                this->key = key;
19                this->item = x;
20                this->next = DPointer<LockFreehash::Node, sizeof(size_t)>();
21            }
22
23            Node* getnext() {
24                bool cMarked[] = { false };
25                bool sMarked[] = { false };
26                Node* succ;
27                Node* entry = this->next.ptr;
28                cMarked[0] = this->next.mark;
29                while (cMarked[0]) {
30                    succ = entry->next.ptr;
31                    sMarked[0] = entry->next.mark;
32                    this->next.ptr = succ;
33                    this->next.mark = sMarked[0];
34                    entry = this->next.ptr;
35                    cMarked[0] = this->next.mark;
36                }
37                return entry;
38            }
39        };
40    public:
41        class Window {
42        public:
43            Node *pred;
44            Node *curr;
45            Window() {
46            }
47            Window(Node *pred, Node *curr) {
48                this->pred = pred;
49                this->curr = curr;
50            }
51        };
52    const static int WORD_SIZE = 24;
53    const static int LO_MASK = 0x00000001;
54    const static int HI_MASK = 0x00800000;
55    const static int MASK = 0x00FFFFFF;
```

```

56     Node *head;
57     LockFreehash() {
58         this->head = new Node(0);
59         Node *tail = new Node(2147483647);
60         while (!head->next.cas(DPointer<LockFreehash::Node,
61             sizeof(size_t)>(tail, 0),NULL));
62     }
63     LockFreehash(Node* e) {
64         this->head = e;
65     }
66     int hashcode(int x) {
67         std::tr1::hash<int> hash_fn;
68         int a = hash_fn(x);
69         return a & MASK;
70     }
71
72     int reverse(int key) {
73         int loMask = LO_MASK;
74         int hiMask = HI_MASK;
75         int result = 0;
76         for (int i = 0; i < WORD_SIZE; i++) {
77             if ((key & loMask) != 0) { // bit set
78                 result |= hiMask;
79             }
80             loMask <<= 1;
81             hiMask >>= 1; // fill with 0 from left
82         }
83         return result;
84     }
85
86     int makeRegularKey(int x) {
87         std::tr1::hash<int> hash_fn;
88         int a = hash_fn(x);
89         int code = a & MASK; // take 3 lowest bytes
90         return reverse(code | HI_MASK);
91     }
92
93     int makeSentinelKey(int key) {
94         return reverse(key & MASK);
95     }
96
97     Window* find(Node* head, int key) {
98         Node* pred = head;
99         Node* curr = head->getnext();
100         while (curr->key < key) {
101             pred = curr;
102             curr = pred->getnext();
103         }
104         return new Window(pred, curr);
105     }
106
107     bool add(int x) {
108         int key = makeRegularKey(x);
109         bool splice;
110         while (true) {
111             Window* window = find(head, key);

```

```

112     Node* pred = window->pred;
113     Node* curr = window->curr;
114     Node* entry = new Node(key, x);
115     entry->next = DPointer < LockFreehash::Node, sizeof(size_t) > (curr,
116         0);
117     splice = pred->next.cas(DPointer<Node, sizeof(size_t)>(entry, 0),
118         curr);
119     if (splice)
120         return true;
121     else
122         continue;
123 }
124 }
125
126 bool remove(int x) {
127     int key = makeRegularKey(x);
128     bool snip;
129     while (true) {
130         Window* window = find(head, key);
131         Node* pred = window->pred;
132         Node* curr = window->curr;
133         if (curr->key != key) {
134             return false;
135         } else {
136             snip = pred->next.cas(DPointer<Node, sizeof(size_t)>(curr, true),
137                 curr);
138             if (snip)
139                 return true;
140             else
141                 continue;
142         }
143     }
144 }
145
146 bool contains(int x) {
147     int key = makeRegularKey(x);
148     Window* window = find(head, key);
149     Node* pred = window->pred;
150     Node* curr = window->curr;
151     return curr->key == key;
152 }
153
154 LockFreehash* getsentinel(int index) {
155     int key = makeSentinelKey(index);
156     bool splice;
157     while (true) {
158         Window* window = find(head, key);
159         Node* pred = window->pred;
160         Node* curr = window->curr;
161         // is the key present?
162         if (curr->key == key) {
163             return new LockFreehash(curr);
164         } else {
165             // splice in new entry
166             Node* entry = new Node(key);
167             entry->next = DPointer < LockFreehash::Node, sizeof(size_t) >
168                 (pred, 0);

```

```

165         splice = pred->next.cas(DPointer<Node, sizeof(size_t)>(entry,
166             false), curr);
167         if (splice) {
168             return new LockFreehash(curr);
169         } else
170             continue;
171     }
172 }
173 };
174
175 vector<LockFreehash*> bucket;
176 int bucketSize;
177 int setSize;
178 const static double THRESHOLD = 4.0;
179 LockFreehash lfh;
180 Lockprogram() {
181 }
182 Lockprogram(int capacity) {
183
184     for (int i = 0; i < capacity; i++) {
185         LockFreehash* temp = new LockFreehash();
186         this->bucket.push_back(temp);
187     }
188     this->bucketSize = 2;
189     this->setSize = 0;
190 }
191
192 LockFreehash* getBucketList(int myBucket) {
193     if (this->bucket[myBucket] == NULL
194         )
195         initializeBucket(myBucket);
196     return this->bucket[myBucket];
197 }
198
199 int getparent(int myBucket) {
200     int parent = this->bucketSize;
201     do {
202         parent = parent >> 1;
203     } while (parent > myBucket);
204     parent = myBucket - parent;
205     return parent;
206 }
207
208 void initializeBucket(int myBucket) {
209     int parent = getparent(myBucket);
210     if (this->bucket[parent] = NULL
211         )
212         initializeBucket(parent);
213     LockFreehash* b = this->bucket[parent]->getsentinel(myBucket);
214     if (b != NULL
215         )
216         this->bucket[myBucket] = b;
217 }
218
219 bool add(int x) {
220     int mybucket = abs(lfh.hashcode(x) % this->bucketSize);

```

```

221     LockFreehash* b = getBucketList(mybucket);
222     if (!b->add(x))
223         return false;
224     int setSizeNow = this->setSize + 1;
225     int bucketSizeNow = this->bucketSize;
226     if (setSizeNow / (double) bucketSizeNow > THRESHOLD)
227         this->bucketSize = 2 * bucketSizeNow;
228     return true;
229 }
230
231 bool remove(int x) {
232     int myBucket = abs(lfh.hashcode(x) % bucketSize);
233     LockFreehash* b = getBucketList(myBucket);
234     if (!b->remove(x))
235         return false;
236     return true;
237 }
238
239 bool contains(int x) {
240     int myBucket = abs(lfh.hashcode(x) % bucketSize);
241     LockFreehash* b = getBucketList(myBucket);
242     return b->contains(x);
243 }
244 }
245 };

```

---



## 2.2.4 List

```
1 class LockFreeList {
2 public:
3     class Node {
4
5     public:
6         int item;
7         int key; //item's hash code
8
9         DPointer<LockFreeList::Node, sizeof(size_t)> next;
10        Node() {
11        }
12        Node(int item1) { // usual constructor
13            this->item = item1;
14            this->key = item1; //instead of hashCode(), we have used the item itself
15                               //as the key.
16            this->next = DPointer<LockFreeList::Node, sizeof(size_t)>();
17        }
18    };
19    Node *head;
20
21    LockFreeList() {
22        this->head = new Node(0);
23        Node *tail = new Node(1000000);
24        while (!head->next.cas(DPointer<LockFreeList::Node, sizeof(size_t)>(tail,
25                                0), NULL));
26    }
27
28    bool add(int item) {
29        int key = item;
30        bool splice;
31        while (true) {
32            Window *window = find(head, key);
33            Node *pred = window->pred, *curr = window->curr;
34            Node *node = new Node(item);
35            node->next = DPointer < LockFreeList::Node, sizeof(size_t) > (curr, 0);
36            if (pred->next.cas(DPointer<LockFreeList::Node, sizeof(size_t)>(node,
37                                0), curr)) {
38                return true;
39            }
40        }
41    }
42
43    bool remove(int item) {
44        int key = item;
45        bool snip;
46        while (true) {
47            Window *window = find(head, key);
48            Node *pred = window->pred, *curr = window->curr;
49            Node *succ = curr->next.ptr;
50            snip = curr->next.cas(DPointer<LockFreeList::Node, sizeof(size_t)>(succ,
51                                    1),succ);
52            if (!snip)
53                continue;
```

```

51         pred->next.cas(DPointer<LockFreeList::Node, sizeof(size_t)>(succ, 0),
52             curr);
53     }
54 }
55
56 bool contains(int item) {
57     int key = item;
58     Window *window = find(head, key);
59     Node *pred = window->pred, *curr = window->curr;
60     return (curr->key == key);
61 }
62
63 class Window {
64 public:
65     Node *pred;
66     Node *curr;
67     Window(Node *pred, Node *curr) {
68         this->pred = pred;
69         this->curr = curr;
70     }
71 };
72
73 LockFreeList::Window* find(Node *head, int key) {
74     Node *pred = NULL, *curr = NULL, *succ = NULL;
75     bool marked[] = { false }; // is curr marked?
76     bool snip;
77     int flag = 0;
78     retry: while (true) {
79         pred = head;
80         curr = pred->next.ptr;
81         while (true) {
82             succ = curr->next.ptr;
83             marked[0] = curr->next.mark;
84
85             while (marked[0]) { // replace curr if marked
86                 snip = pred->next.cas(DPointer<Node, sizeof(size_t)>(succ,
87                     false), curr);
88                 if (!snip) {
89                     goto retry;
90                 }
91                 curr = pred->next.ptr;
92                 succ = curr->next.ptr;
93                 marked[0] = curr->next.mark;
94             }
95             if (curr->key >= key)
96                 return new LockFreeList::Window(pred, curr);
97             pred = curr;
98             curr = succ;
99         }
100     }
101 };

```

---

## 2.2.5 Queue

```
1 class Queue {
2     class Node {
3     public:
4         int value;
5         Node *next;
6
7         Node(int value) {
8             this->value = value;
9             this->next = NULL;
10        }
11    };
12    private:
13        Node *head;
14        Node *tail;
15    public:
16
17        Queue() {
18            Node *sentinel = new Node(-1);
19            this->head = sentinel;
20            this->tail = sentinel;
21        }
22
23    public:
24        void enqueue(int item) {
25            Node *node = new Node(item);
26            Node *last, *next;
27
28            while (true) {
29                last = tail; // read tail
30                next = last->next;
31                if (last == tail) {
32                    if (next == NULL)
33                    {
34                        if
35                            (reinterpret_cast<Node*>(compareAndExchange(reinterpret_cast<volatile
36                                size_t*>(&last->next), reinterpret_cast<size_t>(next),
37                                reinterpret_cast<size_t>(node))) == next) {
38                                compareAndExchange(reinterpret_cast<volatile size_t*>(&tail),
39                                    reinterpret_cast<size_t>(last),
40                                    reinterpret_cast<size_t>(node));
41                                return;
42                            }
43                        } else {
44                            compareAndExchange(reinterpret_cast<volatile size_t*>(&tail),
45                                reinterpret_cast<size_t>(last),
46                                reinterpret_cast<size_t>(next));
47                        }
48                    }
49                }
50            }
51
52            int dequeue() {
53                int c = 0;
54                while (true) {
55                    Node *first = head;
```

```

49     Node *last = tail;
50     Node *next = first->next;
51     if (first == head) { // are they consistent?
52         if (first == last) { // is queue empty or tail falling behind?
53             if (next == NULL || head->value == -1) { // is queue empty?
54                 cout << "\nqueue is empty";
55                 return -1;
56             }
57             compareAndExchange(reinterpret_cast<volatile size_t*>(&tail),
58                                 reinterpret_cast<size_t>(last),
59                                 reinterpret_cast<size_t>(next));
60         } else {
61             int value = next->value;
62             if
63                 (reinterpret_cast<Node*>(compareAndExchange(reinterpret_cast<volatile
64                                                             size_t*>(&head), reinterpret_cast<size_t>(first),
65                                                             reinterpret_cast<size_t>(next)))) == first)
66                 return value;
67         }
68     }
69 }
70 };

```

---

## 2.2.6 Stack

---

```
1 struct Node {
2     int value;
3     struct Node* next;
4 };
5
6 struct Node* top = NULL;
7
8 bool tryPush(Node* node) {
9     Node* oldTop = top;
10    node->next = oldTop;
11    return (reinterpret_cast<struct Node*>(compareAndExchange(
12        reinterpret_cast<volatile size_t*>(&top),
13        reinterpret_cast<size_t>(oldTop), reinterpret_cast<size_t>(node))));
14 }
15
16 void push(int value) {
17     Node* node = (struct Node*) malloc(sizeof(struct Node));
18     while (true) {
19         if (tryPush(node)) {
20             return;
21         }
22     }
23 }
24
25 Node* tryPop() {
26     Node* oldTop = top;
27     if (oldTop == NULL)
28     {
29         cout << "\nEmpty Stack";
30         return NULL;
31     }
32     Node* newTop = oldTop->next;
33     if (reinterpret_cast<struct Node*>(compareAndExchange(reinterpret_cast<volatile size_t*>(&top),
34         reinterpret_cast<size_t>(oldTop), reinterpret_cast<size_t>(newTop))) ==
35         oldTop) {
36         return oldTop;
37     } else {
38         return NULL;
39     }
40 }
41
42 int pop() {
43     while (true) {
44         Node *returnNode = tryPop();
45         if (returnNode != NULL)
46         {
47             return returnNode->value;
48         }
49     }
50 }
```

---

## 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.

---

```
1 #define LOCKTABLESIZE 819412
2
3 int lock[LOCKTABLESIZE];
4 void sort(volatile size_t *list[], volatile size_t oldV[],
5          volatile size_t newV[], int length) {
6     for (int i = 0; i < length; i++) {
7         for (int j = 0; j < length - (i + 1); j++) {
8             if (list[j] > list[j + 1]) {
9                 volatile size_t *temp = list[j + 1];
10                list[j + 1] = list[j];
11                list[j] = temp;
12                volatile size_t tV = oldV[j + 1];
13                oldV[j + 1] = oldV[j];
14                oldV[j] = tV;
15                tV = newV[j + 1];
16                newV[j + 1] = newV[j];
17                newV[j] = tV;
18            }
19        }
20    }
21 }
22
23 void initialiseLockArray() {
24     int i;
25     for (i = 0; i < LOCKTABLESIZE; i++) {
26         int falseValue = 0;
27         lock[i] = falseValue;
28     }
29 }
30
31 void acquireLock(volatile size_t *location) {
32     volatile size_t index = (volatile size_t) location;
33
34     int hashIndex = index % LOCKTABLESIZE;
35     int falseValue = 0;
36     int trueValue = 1;
37     while ((compareAndExchange(
38         reinterpret_cast<volatile size_t*>(&lock[hashIndex]), falseValue,
39         trueValue)) != falseValue) {
40     }
41
42 }
43
44 void releaseLock(volatile size_t *location) {
45     volatile size_t index = ((volatile size_t) location) % LOCKTABLESIZE;
46     int hashIndex = index % LOCKTABLESIZE;
47     int falseValue = 0;
48     lock[hashIndex] = falseValue;
49 }
```

```

50 }
51 void fail(volatile size_t **addressArray, unsigned int index) {
52     unsigned int i;
53     for (i = index; i > 0; i--) {
54         releaseLock(addressArray[i]);
55     }
56 }
57
58 bool mcas(int n, volatile size_t *m0, volatile size_t old0,
59           volatile size_t new0) {
60     initialiseLockArray();
61     int i;
62     volatile size_t *address[1] = { m0 };
63     volatile size_t oldV[1] = { old0 };
64     volatile size_t newV[1] = { new0 };
65     sort(address, oldV, newV, 1);
66
67     for (i = 0; i < n; i++) {
68         acquireLock(address[i]);
69         if ((unsigned long int) *address[i] != (unsigned long int) oldV[i]) {
70             fail(address, i);
71             return false;
72         }
73     }
74
75     if (i == n) {
76         for (i = n - 1; i >= 0; i--) {
77             *address[i] = newV[i];
78             releaseLock(address[i]);
79         }
80         return true;
81     }
82 }
83
84 bool mcas(int n, volatile size_t *m0, volatile size_t *m1, volatile size_t old0,
85           volatile size_t old1, volatile size_t new0, volatile size_t new1) {
86     initialiseLockArray();
87     int i;
88     volatile size_t *address[2] = { m0, m1 };
89     volatile size_t oldV[2] = { old0, old1 };
90     volatile size_t newV[2] = { new0, new1 };
91     sort(address, oldV, newV, 2);
92
93     for (i = 0; i < n; i++) {
94         acquireLock(address[i]);
95         if ((unsigned long int) *address[i] != (unsigned long int) oldV[i]) {
96             fail(address, i);
97             return false;
98         }
99     }
100
101     if (i == n) {
102         for (i = n - 1; i >= 0; i--) {
103             *address[i] = newV[i];
104             releaseLock(address[i]);
105         }
106         return true;

```

```

107     }
108 }
109
110 bool mcas(int n, volatile size_t *m0, volatile size_t *m1, volatile size_t *m2,
111          volatile size_t old0, volatile size_t old1, volatile size_t old2,
112          volatile size_t new0, volatile size_t new1, volatile size_t new2) {
113     initialiseLockArray();
114     int i;
115     volatile size_t *address[3] = { m0, m1, m2 };
116     volatile size_t oldV[3] = { old0, old1, old2 };
117     volatile size_t newV[3] = { new0, new1, new2 };
118     sort(address, oldV, newV, n);
119
120     for (i = 0; i < n; i++) {
121         acquireLock(address[i]);
122         if ((unsigned long int) *address[i] != (unsigned long int) oldV[i]) {
123             fail(address, i);
124             return false;
125         }
126     }
127
128     if (i == n) {
129         for (i = n - 1; i >= 0; i--) {
130             *address[i] = newV[i];
131             releaseLock(address[i]);
132         }
133         return true;
134     }
135 }
136
137 bool mcas(int n, volatile size_t *m0, volatile size_t *m1, volatile size_t *m2,
138          volatile size_t *m3, volatile size_t old0, volatile size_t old1,
139          volatile size_t old2, volatile size_t old3, volatile size_t new0,
140          volatile size_t new1, volatile size_t new2, volatile size_t new3) {
141     initialiseLockArray();
142     int i;
143     volatile size_t *address[4] = { m0, m1, m2, m3 };
144     volatile size_t oldV[4] = { old0, old1, old2, old3 };
145     volatile size_t newV[4] = { new0, new1, new2, new3 };
146     sort(address, oldV, newV, n);
147
148     for (i = 0; i < n; i++) {
149         acquireLock(address[i]);
150         if ((unsigned long int) *address[i] != (unsigned long int) oldV[i]) {
151             fail(address, i);
152             return false;
153         }
154     }
155
156     if (i == n) {
157         for (i = n - 1; i >= 0; i--) {
158             *address[i] = newV[i];
159             releaseLock(address[i]);
160         }
161         return true;
162     }
163 }

```



```

164 }
165
166 bool mcas(int n, volatile size_t *m0, volatile size_t *m1, volatile size_t *m2,
167          volatile size_t *m3, volatile size_t *m4, volatile size_t *m5,
168          volatile size_t old0, volatile size_t old1, volatile size_t old2,
169          volatile size_t old3, volatile size_t old4, volatile size_t old5,
170          volatile size_t new0, volatile size_t new1, volatile size_t new2,
171          volatile size_t new3, volatile size_t new4, volatile size_t new5) {
172     initialiseLockArray();
173     int i;
174     volatile size_t *address[6] = { m0, m1, m2, m3, m4, m5 };
175     volatile size_t oldV[6] = { old0, old1, old2, old3, old4, old5 };
176     volatile size_t newV[6] = { new0, new1, new2, new3, new4, new5 };
177     sort(address, oldV, newV, n);
178
179     for (i = 0; i < n; i++) {
180         acquireLock(address[i]);
181         if ((unsigned long int) *address[i] != (unsigned long int) oldV[i]) {
182             fail(address, i);
183             return false;
184         }
185     }
186 }
187
188 if (i == n) {
189     for (i = n - 1; i >= 0; i--) {
190         *address[i] = newV[i];
191         releaseLock(address[i]);
192     }
193     return true;
194 }
195 }

```

---

### 2.3.1 Binary Search Tree

---

```
1 struct bst {
2     int data;
3     struct bst *left;
4     struct bst *right;
5 };
6 typedef struct bst bst_t;
7 bst_t *root = NULL;
8
9 bst_t *get_new_node(int val) {
10     bst_t *node = (bst_t *) malloc(sizeof(bst_t));
11     node->data = val;
12     node->left = NULL;
13     node->right = NULL;
14     return node;
15 }
16
17 bst_t *insert(bst_t *root, int val) {
18     if (!root)
19         return get_new_node(val);
20     bst_t *prev = NULL, *ptr = root, *last, *temp, *node;
21     node = get_new_node(val);
22     char type = ' ';
23     while (true) {
24         while (ptr) {
25             prev = ptr;
26             if (val < ptr->data) {
27                 ptr = ptr->left;
28                 type = 'l';
29             } else {
30                 ptr = ptr->right;
31                 type = 'r';
32             }
33         }
34         if (type == 'l') {
35             last = prev->left;
36             if(mcas(1, reinterpret_cast<volatile size_t*>(&prev->left),
37                 reinterpret_cast<volatile size_t>(last), reinterpret_cast<volatile
38                 size_t>(node)))
39                 return root;
40         } else {
41             last = prev->right;
42             if(mcas(1, reinterpret_cast<volatile size_t*>(&prev->right),
43                 reinterpret_cast<volatile size_t>(last), reinterpret_cast<volatile
44                 size_t>(node)))
45                 return root;
46         }
47     }
48 }
49
50 int find_minimum_value(bst_t *ptr) {
51     int min = ptr ? ptr->data : 0;
52     while (ptr) {
53         if (ptr->data < min)
54             min = ptr->data;
55         if (ptr->left) {
```

```

52     ptr = ptr->left;
53 } else if (ptr->right) {
54     ptr = ptr->right;
55 } else
56     ptr = NULL;
57 }
58 return min;
59 }
60
61 bst_t *deleter(bst_t *root, int val) {
62     bst_t *prev = NULL, *ptr = root, *last, *present, *temp;
63     char type = ' ';
64     while (ptr) {
65         if (ptr->data == val) {
66             if (!ptr->left && !ptr->right) { // node to be removed has no children's
67                 if (ptr != root && prev) { // delete leaf node
68                     if (type == 'l') {
69                         while (flag1) {
70                             last = prev;
71                             present = ptr;
72                             temp = NULL;
73                             if(mcas(2, reinterpret_cast<volatile size_t*>(&ptr),
74                                 reinterpret_cast<volatile size_t*>(&prev->left),
75                                 reinterpret_cast<volatile size_t*>(present),
76                                 reinterpret_cast<volatile size_t*>(present),
77                                 reinterpret_cast<volatile size_t*>(temp),
78                                 reinterpret_cast<volatile size_t*>(temp)))
79                                 return root;
80                         }
81                     } else {
82                         while (flag1) {
83                             last = prev;
84                             present = ptr;
85                             temp = NULL;
86                             if(mcas(2, reinterpret_cast<volatile size_t*>(&ptr),
87                                 reinterpret_cast<volatile size_t*>(&prev->right),
88                                 reinterpret_cast<volatile size_t*>(present),
89                                 reinterpret_cast<volatile size_t*>(present),
90                                 reinterpret_cast<volatile size_t*>(temp),
91                                 reinterpret_cast<volatile size_t*>(temp)))
92                                 return root;
93                         }
94                     }
95                 } else {
96                     last = root;
97                     temp = NULL;
98                     if(mcas(1, reinterpret_cast<volatile size_t*>(&root),
99                         reinterpret_cast<volatile size_t*>(last),
100                         reinterpret_cast<volatile size_t*>(temp)))
101                         return root;
102                 } // deleted node is root
103             } else if (ptr->left && ptr->right) { // node to be removed has two
104                 children's
105                 ptr->data = find_minimum_value(ptr->right); // find minimum value
106                     from right subtree
107                 val = ptr->data;
108                 prev = ptr;

```

```

95     ptr = ptr->right; // continue from right subtree delete min node
96     type = 'r';
97     continue;
98 } else { // node to be removed has one children
99     if (ptr == root) { // root with one child
100         if (root->left) {
101             last = root;
102             temp = root->left;
103             if(mcas(1, reinterpret_cast<volatile size_t*>(&root),
104                 reinterpret_cast<volatile size_t>(last),
105                 reinterpret_cast<volatile size_t>(temp)))
106                 return root;
107         } else {
108             last = root;
109             temp = root->right;
110             if(mcas(1, reinterpret_cast<volatile size_t*>(&root),
111                 reinterpret_cast<volatile size_t>(last),
112                 reinterpret_cast<volatile size_t>(temp)))
113                 return root;
114         }
115     } else { // subtree with one child
116         if (type == 'l') {
117             if (ptr->left) {
118                 while (flag1) {
119                     present = ptr;
120                     temp = NULL;
121                     if(mcas(2, reinterpret_cast<volatile
122                         size_t*>(&prev->left), reinterpret_cast<volatile
123                         size_t*>(&ptr), reinterpret_cast<volatile
124                         size_t>(present), reinterpret_cast<volatile
125                         size_t>(present), reinterpret_cast<volatile
126                         size_t>(present->left), reinterpret_cast<volatile
127                         size_t>(temp)))
128                         return root;
129                 }
130             } else {
131                 present = ptr;
132                 temp = NULL;
133                 if(mcas(2, reinterpret_cast<volatile size_t*>(&prev->right),
134                     reinterpret_cast<volatile size_t*>(&ptr),
135                     reinterpret_cast<volatile size_t>(present),
136                     reinterpret_cast<volatile size_t>(present),
137                     reinterpret_cast<volatile size_t>(present->right),
138                     reinterpret_cast<volatile size_t>(temp)))
139                     return root;
140             }
141         } else {
142             if (ptr->left) {
143                 last = ptr->left;
144                 present = ptr;
145                 temp = NULL;
146                 if(mcas(2, reinterpret_cast<volatile size_t*>(&prev->left),
147                     reinterpret_cast<volatile size_t*>(&ptr),
148                     reinterpret_cast<volatile size_t>(present),
149                     reinterpret_cast<volatile size_t>(present),
150                     reinterpret_cast<volatile size_t>(last),
151                     reinterpret_cast<volatile size_t>(temp)))

```

```

132         return root;
133     } else {
134         last = ptr->right;
135         present = ptr;
136         temp = NULL;
137         if(mcas(2, reinterpret_cast<volatile size_t*>(&prev->right),
138             reinterpret_cast<volatile size_t*>(&ptr),
139             reinterpret_cast<volatile size_t>(present),
140             reinterpret_cast<volatile size_t>(present),
141             reinterpret_cast<volatile size_t>(last),
142             reinterpret_cast<volatile size_t>(temp)));
143         return root;
144     }
145 }
146 }
147 }
148 prev = ptr;
149 if (val < ptr->data) {
150     ptr = ptr->left;
151     type = 'l';
152 } else {
153     ptr = ptr->right;
154     type = 'r';
155 }
156 }
157 return root;
158 }
159
160 void enq(int value) {
161     root = insert(root, value);
162 }
163
164 void deq(int value) {
165     root = deleter(root, value);
166 }

```

---

### 2.3.2 Doubly Linked List

---

```
1 struct node {
2     int data;
3     struct node *next;
4     struct node *prev;
5 };
6 struct node* head = NULL;
7
8 void insertAfter(struct node* prev_node, int new_data) {
9     struct node* new_node = (struct node*) malloc(sizeof(struct node));
10    new_node->data = new_data;
11    new_node->next = NULL;
12    new_node->prev = NULL;
13    struct node *temp = NULL, *next_l, *prev_l;
14    while (true) {
15        next_l = prev_node->next;
16        prev_l = prev_node;
17
18        if(mcas(4, reinterpret_cast<volatile size_t*>(&new_node->next),
19            reinterpret_cast<volatile size_t*>(&new_node->prev),
20            reinterpret_cast<volatile size_t*>(&prev_node->next),
21            reinterpret_cast<volatile size_t*>(&prev_node->next->prev),
22            reinterpret_cast<volatile size_t*>(temp), reinterpret_cast<volatile
23            size_t*>(temp), reinterpret_cast<volatile size_t*>(next_l),
24            reinterpret_cast<volatile size_t*>(prev_node),
25            reinterpret_cast<volatile size_t*>(next_l), reinterpret_cast<volatile
26            size_t*>(prev_node), reinterpret_cast<volatile size_t*>(new_node),
27            reinterpret_cast<volatile size_t*>(new_node)))
28        return;
29    }
30 }
31
32 void insert(int key) {
33     struct node *prev = head, *curr = prev;
34     while (curr->data < key) {
35         prev = curr;
36         curr = curr->next;
37     }
38     insertAfter(prev, key);
39 }
40
41 void deleteNode(struct node *del) {
42     struct node *prev_l, *next_l;
43     while (true) {
44         prev_l = del->next->prev;
45         next_l = del->prev->next;
46
47         if(mcas(2, reinterpret_cast<volatile size_t*>(&del->next->prev),
48             reinterpret_cast<volatile size_t*>(&del->prev->next),
49             reinterpret_cast<volatile size_t*>(prev_l), reinterpret_cast<volatile
50             size_t*>(next_l), reinterpret_cast<volatile size_t*>(del->prev),
51             reinterpret_cast<volatile size_t*>(del->next)))
52         return;
53     }
54     free(del);
55 }
```

```
43
44 void deleten(int key) {
45     struct node *prev = head, *curr = prev;
46     while (curr->data < key) {
47         prev = curr;
48         curr = curr->next;
49     }
50     deleteNode(curr);
51 }
```

---

### 2.3.3 Hash Set

---

```
1 static vector<struct node* > table;
2 struct node {
3     int data;
4     struct node *next;
5 };
6
7 void linkadd(int num, int index) {
8     bool mcast = false;
9     struct node *temp, *a1, *a2;
10    temp = (struct node *) malloc(sizeof(struct node));
11    temp->data = num;
12    while (!mcast) {
13        if (table[index] == NULL) {
14            table[index] = temp;
15            table[index]->next = NULL;
16            return;
17        } else {
18            a1 = temp->next;
19            a2 = table[index];
20            mcast = mcas(2, reinterpret_cast<volatile size_t*>(&temp->next),
21                        reinterpret_cast<volatile size_t*>(&(table[index])),
22                        reinterpret_cast<volatile size_t*>(a1), reinterpret_cast<volatile
23                        size_t*>(a2), reinterpret_cast<volatile size_t*>(table[index]),
24                        reinterpret_cast<volatile size_t*>(temp));
25        }
26    }
27 }
28
29 bool linkdelete(int num, int index) {
30     struct node *temp, *prev;
31     while (true) {
32         temp = table[index];
33         if (temp != NULL)
34         {
35             if (mcas(1, reinterpret_cast<volatile size_t*>(&(table[index])),
36                     reinterpret_cast<volatile size_t*>(temp), reinterpret_cast<volatile
37                     size_t*>(temp->next)))
38                 return true;
39             } else {
40                 return false;
41             }
42         }
43     }
44 }
45
46 bool linkcontain(int num, int index) {
47     if (table[index] == NULL) {
48         return false;
49     }
50     struct node* r = table[index];
51     while (r != NULL) {
52         if (r->data == num)
53             return true;
54         r = r->next;
55     }
56     return false;
57 }
```



```

50 }
51
52 class CoarseHashSet {
53     int size;
54 public:
55     CoarseHashSet(int capacity) {
56         size = 0;
57         table.clear();
58         for (int i = 0; i < capacity; i++) {
59             struct node *temp = NULL;
60             //temp=(struct node *)malloc(sizeof(struct node));
61             table.push_back(temp);
62         }
63     }
64
65     bool contains(int x) {
66         std::tr1::hash<int> hash_fn;
67         int a = hash_fn(x);
68         bool ans = linkcontain(x, abs(a % table.size()));
69         return ans;
70     }
71
72     void add(int x) {
73         std::tr1::hash<int> hash_fn;
74         int a = hash_fn(x);
75         linkadd(x, abs(a % table.size()));
76         size = size + 1;
77     }
78
79     void remove(int x) {
80         std::tr1::hash<int> hash_fn;
81         int a = hash_fn(x);
82         bool result = linkdelete(x, abs(a % table.size()));
83         size = result ? size - 1 : size;
84     }
85 };

```

---

### 2.3.4 List

---

```
1 class List {
2 public:
3     class Node {
4     public:
5         int item;
6         int key;
7         Node *next;
8         Node(int item) {
9             this->item = item;
10            this->key = item;
11        }
12    };
13    Node *head;
14    Node *tail;
15
16    List() {
17        head = new Node(-2147483648); // Add sentinels to start and end
18        tail = new Node(2147483647);
19        head->next = this->tail;
20    }
21
22    bool add(int item) {
23        Node *pred, *curr, *last, *temp;
24        int key = item;
25        bool flag, mcast = false;
26        Node *node = new Node(item);
27        while (true) {
28            pred = head;
29            curr = pred->next;
30            while (curr->key < key) {
31                pred = curr;
32                curr = curr->next;
33            }
34            last = curr;
35            temp = NULL;
36            mcast = mcast(2, reinterpret_cast<volatile size_t*>(&pred->next),
37                          reinterpret_cast<volatile size_t*>(&node->next),
38                          reinterpret_cast<volatile size_t>(last), reinterpret_cast<volatile
39                          size_t>(temp), reinterpret_cast<volatile size_t>(node),
40                          reinterpret_cast<volatile size_t>(last));
41            if (mcast) {
42                flag = true;
43                return flag;
44            }
45        }
46    }
47
48    bool remove(int item) {
49        Node *pred, *curr, *last, *temp, *prev;
50        int key = item;
51        while (true) {
52            pred = this->head;
53            curr = pred->next;
54            while (curr->key < key) {
55                pred = curr;
```

```

52         curr = curr->next;
53     }
54     last = curr->next;
55     prev = pred->next;
56     mcas(1, reinterpret_cast<volatile size_t*>(&pred->next),
          reinterpret_cast<volatile size_t>(prev), reinterpret_cast<volatile
          size_t>(last));
57     return true;
58 }
59 }
60
61 bool contains(int item) {
62     Node *pred, *curr, *last, *prev, *temp;
63     int key = item;
64     pred = head;
65     curr = pred->next;
66     while (curr->key < key) {
67         prev = pred;
68         last = curr;
69         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));
70     }
71     return (key == curr->key);
72 }
73
74 int size() {
75     Node *pred = head;
76     int l = 0;
77     while (pred != tail) {
78         l++;
79         pred = pred->next;
80     }
81     return l;
82 }
83 };

```

---

### 2.3.5 Queue

---

```
1 struct node
2 {
3     int data;
4     node *next;
5 }*front = NULL, *rear = NULL, *p = NULL, *np = NULL, *last = NULL, *first =
    NULL, *next1 = NULL;
6
7 void enqueue(int x) {
8     bool result = false;
9     np = new node;
10    np->data = x;
11    np->next = NULL;
12    while (!result) {
13        last = rear;
14        first = front;
15        if (front == NULL) {
16            front = rear = np;
17        } else {
18            next1 = last->next;
19            result = mcas(2, reinterpret_cast<volatile size_t*>(&rear->next),
20                        reinterpret_cast<volatile size_t*>(&rear),
21                        reinterpret_cast<volatile size_t>(next1), reinterpret_cast<volatile
22                        size_t>(last), reinterpret_cast<volatile size_t>(np),
23                        reinterpret_cast<volatile size_t>(np));
24        }
25    }
26 }
27
28 void dequeue() {
29     bool result = false;
30     while (true) {
31         if (front == NULL) {
32             return;
33         } else {
34             p = front;
35             result = mcas(1, reinterpret_cast<volatile size_t*>(&front),
36                         reinterpret_cast<volatile size_t>(p), reinterpret_cast<volatile
37                         size_t>(front->next));
38             if (result)
39                 return;
40         }
41     }
42 }
```

---

### 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.

---

```
1  class stack {
2  public:
3      class Node {
4      public:
5          int item;
6          Node *next;
7          Node(int item) {
8              this->item = item;
9              this->next = NULL;
10         }
11     };
12     Node *top;
13
14     void push(int value) {
15         bool flag = false;
16         Node *node = new Node(value);
17         Node *topd, *temp = NULL;
18         while (true) {
19             topd = top;
20             flag = mcas(2, reinterpret_cast<volatile size_t*>(&node->next),
21                         reinterpret_cast<volatile size_t*>(&top), reinterpret_cast<volatile
22                         size_t>(temp), reinterpret_cast<volatile size_t>(topd),
23                         reinterpret_cast<volatile size_t>(topd), reinterpret_cast<volatile
24                         size_t>(node));
25             if (flag) {
26                 return;
27             }
28         }
29     }
30
31     void pop() {
32         int t;
33         Node *topd;
34         int temp;
35         bool result = false;
36         while (true) {
37             if (top != NULL) {
38                 t = top->item;
39                 topd = top;
40                 result = mcas(1, reinterpret_cast<volatile size_t*>(&top),
41                             reinterpret_cast<volatile size_t>(topd),
42                             reinterpret_cast<volatile size_t>(top->next));
43                 if (result) {
44                     return;
45                 }
46             } else {
47                 return;
48             }
49         }
50     }
51 };
52
```

---

## 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.