Concurrent AVL Trees

Peter Zijlstra <peter@xlnt-software.com>   Marijn Meijles <marijn@xlnt-software.com>

February 25, 2002

Abstract

This paper describes a concurrent version of the well known AVL search tree. The intention is to illustrate the locking scheme devised. An intimate knowledge of AVL trees is assumed [1]. Also one is assumed to be familiar with C and multi-threading. We will first refine the goals, followed by a discussion of the basic data structures and locking semantics. After that we will elaborate on the various parts of the algorithm and finalize the locking semantics.

Goals

This scheme has two goals:

1. Maximize concurrency in order to minimize latency and boost performance on multiple execution units, and
2. Minimize extra overhead.

Let's further refine the 'maximize concurrency' goal:

- Non-modifying operations don't hinder each other. This goal is quite trivial as reads will be far more common in most applications.
- Modifying operations try to lock as few items as possible and keep those locks as short as possible.
- Try to be transparent for non-modifying operations wherever possible, which overlaps the previous goal, but is certainly not the same .

Data structure

There are two different ways to implement the data structure of a standard AVL. One is with parent pointers, the other one keeps track of the tree descent with a stack. We'll use the stack based version, for these reasons:

- This keeps the nodes connectedness to a minimun; only downward; which helps achieve the second goal of concurrent data structures as we have less bookkeeping to do.
- One would need to do the bookkeeping anyway in order to keep track of the nodes that were locked during the descent. This may sound strange, but we'll introduce a partial unlock from top to bottom later on.

```
typedef struct avl_node_s {
  struct avl_node_s  * avl_link[2];
  int                  avl_balance;
  void               * avl_item;
} avl_node_t;

typedef struct avl_tree_s {
  avl_node_t           avl_root;
  int                  avl_count;
  int                  (*avl_compare)( void *, void *);
  void                 (*avl_freeitem)( void *);
} avl_tree_t;

typedef struct avl_stack_elem_s {
  avl_node_t         * se_node;
  int                  se_dir;
} avl_stack_elem_t;

typedef struct avl_stack_t {
  avl_stack_elem_t   s_elems[AVL_MAX_STACK];
  int                  s_size;
} avl_stack_t;
```

These data structs are what would be needed to build a regular, stack based AVL implementation.

Locking requirements

Obviously we need to introduce some changes to this model in order to introduce concurrency. Locking the whole tree for all modifying operations is the quick way, but surely this does not achieve high concurrency. Analysis of the concurrency goals leads us to propose that multistate locking of the nodes would be needed in order to provide maximum concurrency. For if only one state were available, that state would need to be exlusive, and therefore our first goal wouldn't be met. So we'll need a non-exclusive state (a read state), and one or more (semi) exlusive states. A standard read/write lock wouldn't do either, as we distinguish between two 'write modes'. We'll modify the `avl_node` structure to include a locking field:

```
typedef struct avl_node_s {
/*  ... as before ...     */
  avl_lock_t           avl_lock;
} avl_node_t;
```

We'll specify the lock later on, for now it's sufficient to know that it supports a read lock operation `rlock()`.

Search

Having found that we need a non-exclusive state for non-modyfing operations we'll see if that is enough to implement the most fundamental and only non-modifying operation of AVL trees: search. We employ simple ladder-locking to traverse the tree.

```
int avl_search( avl_tree_t * tree, void * item,
                void ** found) {
  int rc = 0, cmp;
  avl_node_t * p;

  assert( tree && item && found);

  rlock( (p = &tree->avl_root));
  for ( cmp = -1, *found = NULL;;
        cmp = tree->avl_compare( p->avl_item,
                                 item)) {
    avl_node_t * q;

    if ( cmp == 0) {
      *found = p->avl_item;
      break;
    }
    if ( !(q = p->avl_link[ cmp > 0])) {
      rc = -1;
      break;
    }
    rlock( q);
    runlock( p);
    p = q;
  }
  runlock( p);

  return rc;
}
```

This seems almost too easy, and it is. The experienced programmer will already have noticed our first race condition. The problem is that when the function returns success (rc = 0), `*found` may be an invalid pointer. This is due to the fact that as soon as we release the `avl_node` that refers to our `*found` item, nothing stops a delete from freeing our item. We clearly need to insure that our item stays valid. Simply keeping the lock on the corresponding `avl_node` would produce deadlocks so we need another mechanism. We'll take the easy road out, and leave it to the users of our data structure to insure that all goes well, but in order for them to do so we need to provide them with some support. We'll define a callback function which may lock the item.

```
typedef struct avl_tree_s {
/*   ... as before ...  */
  int    (*avl_lockitem)( void *, int);
} avl_tree_t;
```

The `int` argument could be used to enumerate the kind of itemlock that is required, think of items that have r/w locks. Now we need to modify `search()` to include a call to it before `runlock()`:

```
int avl_search( avl_tree_t * tree, void * item,
                void ** found, int lock) {

/*  ... as before ...  */
  if ( *found && tree->avl_lockitem)
    rc = tree->avl_lockitem( *found, lock);
  runlock( p);

  return rc;
}
```

Now the item can get locked/trylocked (due to the rc assignment) or whatever the user needs to do to insure that `*found` is valid upon success.

Insert

Now for the hard part: modifying operations. We'll start with insert, which is the easiest of the two non-modifying operations supported by search trees; delete being the other one.

Let's recapitulate the algorithm:

- Descend the tree to find the parent of the new entry and keep track of visited nodes;
- Add the entry;
- Unwind the descent stack and rebalance the nodes; break the loop when `abs(avl_balance) != 0`, because:
  - `avl_balance == 0`, subtree depth didn't change;
  - `abs(avl_balance) == 2`, a reorder can yield the first result.

If this is an unfamiliar view of the algorithm to you we advise you to review it carefully.

Locking requirements - part II

From the algorithm's point of view there are two kinds of nodes that we might want to look at with respect to locking: the stack nodes and reorder nodes.

The stack nodes are rather difficult. Let us first look at how a modifying operation moves in the tree. The first movement is in a downward direction; finding the parent for our new node. Next we move up; unwinding the stack and rebalancing the nodes. And finaly another downward motion is used when we reorder the tree (this might occur more than once when we have done a delete). So we have a down-up-down movement pattern, which can 'grab' into a concurrent reorder and cause a deadlock. To avoid this we have to make sure only downward motion is used as the default 'locking order'. This is achieved by holding locks on the stack nodes when we descend the tree in search of our lucky parent. Next we need to investigate what kind of lock would be most efficient. It needs to be exclusive in the sense that no other concurrent modify operation can get it, as this causes deadlocks. Readers however will only lock in a downward fashion and so they don't need to be excluded from the lock.

On the contrary, with the reorder nodes it is easy to see that they need locking. The reorder operation changes the tree structure and that's why they require an exclusive lock.

We have found that we need a lock with 3 states, corresponding to: reader nodes, stack nodes and reorder nodes. Each has specific properties, let's list them:

- *Reader*: a non exclusive lock which tells other threads that this node is being looked at and should not be modified.
- *Stack*: a stack node can only be locked by one thread to modify it, which means it excludes other stacks and reorders. Readers however can go through.
- *Reorder*: the tree structure is being changed. Nothing should interrupt.

This set of requirements yields the following state compatibility matrix:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Where state 1 corresponds to the read-state, state 2 to the 'stack' state and state 3 to the fully exclusive reorder state.

We've come a long way in figuring out the locking model for a concurrent AVL tree. However, there is one more crucial concurrency issue which we need to address. In the current state modifying operations still exclude each other because they all want the rootnode locked on their stack.

When you carefully study the algorithm you will see that there is no need to keep the whole descent stack locked. It will suffice to hold only the part downwards from the parent of the last non-null balance node. This presents a loophole we'll employ to make modifying operations concurrent. The worst case is that the rootnode is still held on the stack, but with a sufficiently deep tree this will hardly ever occur. If you're wondering, delete also allows such a trick.

There are also other methods to increase concurrency in AVL trees, most notably the relaxed balancing method [2]. We did not choose this method because of the following reasons:

- Relaxed balancing requires more cpu power because you have to scan the whole tree in search of unbalanced nodes. This is particularly true with large trees, which we intend to use.
- Relaxed balancing is most useful in case of update bursts which will generally not occur in our application.

General statelocks

Most threading libraries don't provide such elaborate locks as the one we need. As building one yourself isn't that hard we'll just give the code. If this was a paper on C++ generic programming we'd show you a template class of the form:

```
template <
  const int N,
  typename ThreadingModel
> class statelock;
```

But it's not, so we'll provide you with a little less generic code that uses the POSIX thread interface (minus the pthread_ prefix) and is hard coded to have 3 states, but is otherwise perfectly general.

As a little bonus, this implementation also allows for trying a lock operation. This isn't needed here, but some might find it handy.

Let us list the requirements:

- Multi-state (3 actually),
- User supplied compatibility matrix.
- Atomic state switching (we need to change stack into reoder nodes).

```
typedef struct trilock_s {
  cond_t              cond;
  mutex_t             mutex;
  volatile int        state[3];
  const int           (* tr)[3][3];
} trilock_t;

int trilock( trilock_t * lock,
             int from, int to, int try) {
  mutex_lock( &lock->mutex);
  if ( from--)
    assert( lock->state[from] > 0);

  if ( to--) {
    int compat;
    do {
      int i;
      for ( compat = 2, i = 0; i < 3; i++) {
        int s = lock->state[i];
        if ( i == from) s--;
        if ( s && !(*lock->tr)[i][to]) {
          compat = 0;
          break;
        }
      }
      if ( try && !compat) {
        mutex_unlock( &lock->mutex);
        return EBUSY;
      }
    } while ( !compat &&
             !cond_wait( &lock->cond, &lock->mutex));
    lock->state[to]++;
  }

  if ( from >= 0) {
    lock->state[from]--;
    cond_broadcast( &lock->cond);
  }

  mutex_unlock( &lock->mutex);
  return 0;
}
```

For the speed freaks out there: this is indeed overkill for a trilock with a static compatibility matrix. However we're only here to show how to do it, not how to implement it ;). And it's not that bad anyway.

Overweight

For the people who feel this lock is way to heavy, say kernel developers, there is another way in which the locking can be done. Consider the lock compatibility matrix:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

This lock removes the atomic state switch requirement, and makes states 2 and 3 into a single multi stage state, thereby allowing it to be implemented using a lock implementation that only supports lock/unlock operations, in contrast to lock/-switch/unlock. There is one extra requirement though, being that you always have to grab state 2 before state 3. This is because state 3 isn't independent anymore.

We don't use this matrix as we think the locking semantics are less natural, and may be confusing. But it's an interesting sidestep.

Creating the stack

Creating the stack is a very straightforward process. There are a few noteworthy points with this implementation though. The last compare result is stored in `*cmp`, this result is always needed. Furthermore, the returned node might not be a stack element, because when there is an exact match the direction is not properly defined nor is the node a parent in that case. As agreed '2' locks are held for all elements traversed. The partial stack trick is hidden in the 'method' call. This is done because it's not universal for all modifying operations.

```
#define NULL_STACK   (avl_stack_t){ .s_size = 0 }
#define PUSH( s, n, d)                 \
  (s)->s_elems[(s)->s_size++] =       \
    (avl_stack_elem_t){ (n), (d) }
#define POP( s)       (s)->s_size--
#define SIZE( s)      (s)->s_size
#define LAST( s)      ((s)->s_size - 1)
#define ELEM( s, i)   (s)->s_elems[(i)]
#define NODE( s, i)   ELEM( s, i).se_node
#define DIR( s, i)    ELEM( s, i).se_dir

typedef void (*method_t)( avl_node_t *,
                          avl_stack_t *);

static inline avl_node_t *
avl_stack_build( avl_tree_t * tree,
                 avl_stack_t * stack,
                 void * item, int * cmp,
                 method_t method) {
  avl_node_t * p = &tree->avl_root;

  trilock( &p->avl_lock, 0, 2, 0);
  for ( *cmp = -1;;
        *cmp = tree->avl_compare( p->avl_item,
                                  item)) {
    int dir = *cmp > 0;
    avl_node_t * q;
    if ( *cmp == 0) break;
    method( p, stack);
    PUSH( stack, p, dir);
    if ( !(q = p->avl_link[dir])) break;
    trilock( &q->avl_lock, 0, 2, 0);
    p = q;
  }

  /* if *cmp == 0, p !e stack */
  return p;
}

static inline void
avl_stack_unlock( avl_tree_t * tree,
                  avl_stack_t * stack) {
  int i;
  for ( i = 0; i < SIZE( stack); i++)
    trilock( &NODE( stack, i)->avl_lock, 2, 0, 0);
  SIZE( stack) = 0;
}
```

The unlock function unlocks top-bottom because the top nodes have the highest probability of pending locks.

Reordering

Reordering the tree is the most complex part of implementing balanced trees. To keep it generic and readable we've split things up.

The `avl_rotate` function performs a very elemental tree rotate followed by an AVL balance factor recalculation. If this is unfamiliar to you we refer you to your favorite book on balanced trees.

```
#define BAL( x) ((x)->avl_balance)

static inline avl_node_t *
avl_rotate( avl_node_t * x, int dir, const int lock) {
  avl_node_t * y = x->avl_link[dir];

  if ( lock) trilock( &y->avl_lock, 0, 3, 0);

  x->avl_link[ dir] = y->avl_link[!dir];
  y->avl_link[!dir] = x;

  if ( dir) {
    BAL(x) = (BAL(x)-1) - ((BAL(y)>0) ? BAL(y) : 0);
    BAL(y) = (BAL(y)-1) + ((BAL(x)<0) ? BAL(x) : 0);
  } else {
    BAL(x) = (BAL(x)+1) - ((BAL(y)<0) ? BAL(y) : 0);
    BAL(y) = (BAL(y)+1) + ((BAL(x)>0) ? BAL(x) : 0);
  }

  trilock( &y->avl_lock, 3, 0, 0);

  return y;
}
```

The `avl_reorder` function performs the more elaborate double rotate when the balance factors are asymetrical. The division takes care of detecting whether the two balance factors are of opposite sign (the multiplication is mathematically redundant but serves to correct for integer rounding errors; eg. -0.5 becomes 0).

```
static inline avl_node_t *
avl_reorder( avl_node_t * x, int * bal) {
  int dir = x->avl_balance > 0;
  avl_node_t * p, * y = x->avl_link[dir];

  trilock( &x->avl_lock, 2, 3, 0);
  trilock( &y->avl_lock, 0, 3, 0);

  if ( bal) *bal = y->avl_balance;

  if ( 2*y->avl_balance / x->avl_balance < 0)
    x->avl_link[dir] = avl_rotate( y, !dir, 1);

  p = avl_rotate( x, dir, 0);

  trilock( &x->avl_lock, 3, 0, 0);

  return p;
}
```

The locking is not terrible difficult but very subtle. The transition from state '2' to state '3' locks should go downward, otherwise we may conflict with readers and cause a deadlock.

Insert implemented

As we now have all the ingredients to build modifying operations we'll go on and do just that. The `avl_insert_stack` function is the method provided to `avl_stack_build` and takes care of unlocking all the unnecessary stack nodes.

```
static inline void
avl_insert_stack( avl_node_t * node,
                  avl_stack_t * stack) {
  int i, s = SIZE( stack);
  if ( !(node->avl_balance && s))
      return;
  for ( s--, i = 0; i < s; i++)
    trilock( &NODE( stack, i)->avl_lock, 2, 0, 0);
  ELEM( stack, 0) = ELEM( stack, s);
  SIZE( stack) = 1;
}
```

This is our actual `avl_insert` function. It it shorter than any other AVL insert I've ever come across but one could argue that we've hidden all the complex parts in some rather general functions. All that remains is the actual algorithm.

```
int avl_insert( avl_tree_t * tree, void * item) {
  avl_stack_t stack = NULL_STACK;
  avl_node_t * p, * n;
  int cmp;

  p = avl_stack_build( tree, &stack,
                       item, &cmp,
                       avl_insert_stack);
  if ( !cmp) {
    trilock( &p->avl_lock, 2, 0, 0);
    avl_stack_unlock( tree, &stack);
    return -1;
  }

  n = avl_new_node( tree, item);
  if ( !n) {
    avl_stack_unlock( tree, &stack);
    return -1;
  }

  trilock( &p->avl_lock, 2, 3, 0);
  p->avl_link[ cmp > 0] = n;
  trilock( &p->avl_lock, 3, 2, 0);

  tree->avl_count++;

  for ( ; SIZE( &stack) > 0; POP( &stack)) {
    p = NODE( &stack, LAST( &stack));
    if ( p == &tree->avl_root) break;
    p->avl_balance +=
      DIR( &stack, LAST( &stack)) ? 1 : -1;
    switch( abs( p->avl_balance)) {
      case 0: goto double_break;
      case 1: break;
      case 2: POP( &stack);
          n = NODE( &stack, LAST( &stack));
          trilock( &n->avl_lock, 2, 3, 0);
          n->avl_link[ DIR( &stack, LAST( &stack))] =
            avl_reorder( p);
          trilock( &n->avl_lock, 3, 0, 0);
          POP( &stack);
          goto double_break;
      default: assert( 0);
    }
    trilock( &p->avl_lock, 2, 0, 0);
  }
double_break:

  avl_stack_unlock( tree, &stack);
  return 0;
}
```

Some people may not like the use of goto, but we think that in some cases it makes the code clearer. Note that reorder unlocks upon return, saving us an extra call to trilock. Tight bookkeeping with the stack is imperative because double unlocking of elements would lead to disaster. Also note that the balance update is done under a '2' lock. This is possible because readers don't care about the balance factors.

Delete implemented

Now for the big trick: 'delete'. This is a notorious hard operation. The stack method greatly resembles the one for insert, but it is a bit weaker, not as tight. To make this function 'perfect' you would have to do a fake reorder pass to precalculate the balance factors. This can be done, but the extra cpu cycles will probably outweigh the improved concurrency in terms of overall speed. So we stuck to the unconditional termination condition and got this:

```
static inline void
avl_delete_stack( avl_node_t * node,
                  avl_stack_t * stack) {
  int i, s = SIZE( stack);
  if ( !(!node->avl_balance && s))
    return;
  for ( s--, i = 0; i < s; i++)
    trilock( &NODE( stack, i)->avl_lock, 2, 0, 0);
  ELEM( stack, 0) = ELEM( stack, s);
  SIZE( stack) = 1;
}
```

We saved the best for last. This implementation of delete first tries all the easy ways to find a replacement. Failing this we pick the deepest subtree. This way we avoid a (potential) reorder. Again locking isn't all that difficult but there are some subtle things to be aware of. Remember that `avl_stack_build` doesn't put it's return node on the stack when `cmp == 0`.

```
int avl_delete( avl_tree_t * tree, void * item,
                const int freeit) {
  avl_stack_t stack = NULL_STACK;
  avl_node_t * d, * r, * s;
  avl_stack_elem_t pe;
  int cmp, i, j;

  d = avl_stack_build( tree, &stack,
                       item, &cmp,
                       avl_delete_stack);

  if ( cmp) {
    avl_stack_unlock( tree, &stack);
    return -1;
  }

  pe = ELEM( &stack, LAST( &stack));
  trilock( &pe.se_node->avl_lock, 2, 3, 0);

  /* check for one/no child(ren) situation */
  for ( i = 0; i < 2; i++) {
    if ( d->avl_link[i] == NULL) {
      pe.se_node->avl_link[pe.se_dir] =
        d->avl_link[!i];
      goto rebalance;
    }
  }

  /* check asymetric no child */
  for ( i = 0; i < 2; i++) {
    avl_node_t * x = d->avl_link[i];
    trilock( &x->avl_lock, 0, 3, 0);
    if ( x->avl_link[i] == NULL) {
      x->avl_link[!i] = d->avl_link[!i];
      trilock( &x->avl_lock, 3, 2, 0);
      x->avl_balance = d->avl_balance;
      PUSH( &stack, x, i);
      pe.se_node->avl_link[pe.se_dir] = x;
      goto rebalance;
    }
    trilock( &x->avl_lock, 3, 0, 0);
  }

  /* do it the hard way; avoid rebalance by picking
     the deepest side */
  i = d->avl_balance > 0;
  r = d->avl_link[i];
  j = SIZE( &stack)++;
  trilock( &r->avl_lock, 0, 2, 0);
  for (;;) {
    PUSH( &stack, r, !i);
    s = r->avl_link[!i];
    trilock( &s->avl_lock, 0, 2, 0);
    if ( s->avl_link[!i] == NULL) break;
    r = s;
  }

  trilock( &r->avl_lock, 2, 3, 0);
  trilock( &s->avl_lock, 2, 3, 0);

  s->avl_link[!i] = d->avl_link[!i];
  r->avl_link[!i] = s->avl_link[ i];
  s->avl_link[ i] = d->avl_link[ i];
  s->avl_balance  = d->avl_balance;

  trilock( &s->avl_lock, 3, 2, 0);
  trilock( &r->avl_lock, 3, 2, 0);

  pe.se_node->avl_link[pe.se_dir] = s;
  ELEM( &stack, j) = (avl_stack_elem_t){ s, i };

rebalance:
  trilock( &pe.se_node->avl_lock, 3, 2, 0);

  tree->avl_count--;

  for ( ; SIZE( &stack) > 0; POP( &stack)) {
    s = NODE( &stack, LAST( &stack));
    if ( s == &tree->avl_root) break;
    s->avl_balance -=
      DIR( &stack, LAST( &stack)) ? 1 : -1;
    switch( abs( s->avl_balance)) {
      case 0: trilock( &s->avl_lock, 2, 0, 0);
          break;
      case 1: goto double_break;
      case 2: POP( &stack);
          r = NODE( &stack, LAST( &stack));
          trilock( &r->avl_lock, 2, 3, 0);
          r->avl_link[ DIR( &stack, LAST( &stack))] =
            avl_reorder( s, &i);
          trilock( &r->avl_lock, 3, 0, 0);
          POP( &stack);
          if ( !i) goto double_break;
          break;
      default: assert( 0);
    }
  }
double_break:
```

```
  avl_stack_unlock( tree, &stack);

  trilock( &d->avl_lock, 2, 3, 0);
  if ( freeit && tree->avl_freeitem)
    tree->avl_freeitem( d->avl_item);
  avl_node_delete( d);

  return 0;
}
```

Another thing to note is that the deleted node 'd' is not '3' locked until the very end. This has a very good reason, if one would '3' lock 'd' before it was removed from the tree (ie. before the rebalance label) there could be pending readers. These readers would unblock after the node was deleted which would lead to a non-consistent view of the tree. However, not '3' locking it before deletion could mean that a reader is still using the node upon deletion, which is also not a very healthy situation. This is why `avl_node_delete` requires the argument to be '3' locked.

References

[1] ftp://cscw.msu.edu/blp/avl-2001.11.10.text

[2] http://www.imada.ou.dk/~kslarsen/RelBal/