

无锁并发二叉搜索树的实现

刘少东*, 邢永康, 刘 恒

(重庆大学 计算机学院, 重庆 400044)

(* 通信作者电子邮箱 liushaodong1989@gmail.com)

摘 要: 针对异步共享内存模型下的并发搜索二叉树(BST)数据结构, 提出了一种新的无锁实现方法。通过一种有效的节点重用策略, 使得删除操作是无等待的, 插入操作是无锁的。实验数据表明, 该数据结构是高度可扩展的而且在高负载下能提供很高的吞吐量。

关键词: 无锁搜索二叉树; 无锁; 无等待; 可扩展; 高吞吐量

中图分类号: TP311.12 **文献标志码:** A

Lock-free implementation of concurrent binary search tree

LIU Shao-dong*, XING Yong-kang, LIU Heng

(College of Computer Science, Chongqing University, Chongqing 400044, China)

Abstract: A new scheme for unlocking implementation of concurrent Binary Search Tree (BST) based on asynchronous shared memory systems was provided in this paper. This scheme possessed two outstanding advantages: The deletion is wait-free, and the insertion is lock-free. The experimental results show that this scheme is highly scalable and can produce high throughputs under heavy load.

Key words: unlocking binary search tree; lock-free; wait-free; scalable; high parallel

0 引言

并发的搜索二叉树(Binary Search Tree, BST)很容易通过锁来实现: Guibas 等^[1]通过解配对再重新平衡更新操作实现了一个平衡的 BST; Nurmi 等^[2]则使用了 chromatic tree, 它是一个叶子导向的放宽了平衡条件的红黑树; Boyar 等^[3]则改进了基于 chromatic tree 的方案。上面的实现都有一个共同的缺点: 当一个线程尝试去修改某个节点时, 它必须锁住该节点附近的一定数目的节点, 这便导致了其他尝试去修改这些节点的线程将会被阻塞。Barnes^[4]改进了上述的方法, 通过一个方法将基于锁的改进成非阻塞的实现。他的实现中, 一个操作会在获取锁之前写入一些信息, 然后其他的线程会帮助这个操作来获取这个锁住的节点, 这种方式下锁最终就会被释放。尽管这种方法可以提高性能, 但是可能存在某个帮助线程在自己到达目的节点之前反复地去帮助其他和它在同一个路径上的线程, 这便会引起该线程较大的时延。Ellen 等^[5]实现了一个叶子导向的 BST 而且支持无锁的插入和删除操作。他对 BST 的条件做了一些放宽: 所有的键值都只存储在叶子节点上, 而且每个内部的节点都有两个子节点, 内部的节点都是导向节点。尽管它是无锁的, 但实现过于复杂: 插入操作需要 3 个 CAS(Compare And Set, CAS)原子操作步骤, 删除操作则需要 4 个。Bronson 等^[6]也实现的是平衡的 BST, 使用一个部分的 external tree 来替代一个整个 external tree 而获得改进。他们的实现中, 一条路径上的节点可以通过 put(k, v)方法而转换到另一个常规节点上, 但是这种重用节点策略过于严格, 导致重用率比较低。

Shavit^[7]的观点说明设计并发数据结构的时候可以放宽一些条件(例如一致性条件^[8])等)来提高并发数据结构的性

能。本文的节点重用策略即是基于该思想的, 该重用策略可以高效地重用逻辑删除留下的节点, 从而提高性能。

1 带节点重用的并发搜索二叉树

算法定义了三种操作: 查找、插入和删除。为方便实验, 测试的 BST 里面存放的 node 中只存了键值。

1.1 节点定义

节点总是处于三种状态中的某一种, 用 status 来标识: 有效的节点标识为“CLEAN”, 逻辑删除节点的标识为“DELETED”, 无效的节点标识为“CHANGING”。节点的 long number 域是节点重用过程中的一个标识量。一个 key 为 k 的节点存在于 BST 中则说明在 BST 中存在一个有效的而且键值为 k 的节点。节点定义如下:

```
Type Definition. NodeStatus;
Enum nodeStatus { CLEAN, DELETED, CHANGING };
STRUCT Node
    volatile node * left;
    volatile node * right;
    volatile nodeStatus status;
    volatile long logNumber;
    int key;
END
```

1.2 节点重用策略

Bronson 等^[6]的重用逻辑删除节点的策略仅仅当原来删除节点的 key 和待插入节点的 key 相同时才会重用。实际上, 一个 insert(k)只要满足以下条件都可以使用节点重用:

条件 1 当 k 大于某个逻辑删除节点的左子树节点中的最大 key 值而且小于这个逻辑删除节点右子树节点中的最小 key 值。只考虑条件 1 的两种子情况进行节点重用。

收稿日期: 2012-03-21; 修回日期: 2011-05-27。

作者简介: 刘少东(1989-), 男, 安徽池州人, 硕士研究生, 主要研究方向: 并发数据结构; 邢永康(1964-), 男, 重庆人, 副教授, 博士, 主要研究方向: 分布式系统、数据挖掘; 刘恒(1988-), 男, 江苏宿迁人, 硕士研究生, 主要研究方向: 并发数据结构。

条件2 k 正好等于一个逻辑删除节点的 key 值而无论该逻辑删除节点是内部节点还是叶子节点,这种重用策略和 Bronson 等^[6]中的策略是一样的。

条件3 k 不等于情况1)中的逻辑删除节点的 k 值,而且这个逻辑删除的节点不是叶子节点。

定义1 对于一个 insert(k) 操作,如果 k 符合条件1,那么条件1中描述的逻辑删除节点被称为本次 insert 操作的关键节点(Critical Node, CN);如果不满足条件1,那么离它应该插入的树中位置最近的逻辑删除节点也被称为关键节点。

对于满足条件2的 insert(k) 操作,重用是很简单的:只需要将关键节点的状态从“DELETED”改成“CLEAN”。由于现在硬件水平的限制,一个 CAS 操作不能原子地同时更新一个节点的 key 域和 status 域,因此加上了一个“CHANGING”作为一个中间状态。满足条件2下的 insert(k) 操作步骤如下:首先使用一个 CAS 操作将节点的状态从“DELETED”改变为“CHANGING”,如果该 CAS 操作成功的话则转到步骤2,如果该操作失败则放弃本次尝试;验证前面使用的关键节点 key 值从变为关键节点后到现在正在进行的操作的瞬间时刻没有发生改变,如果没有发生改变则将 status 改成“CLEAN”;否则的话放弃本次尝试。

但是条件3中的节点重用就不那么直接了,而且容易出现错误,如图1所示。

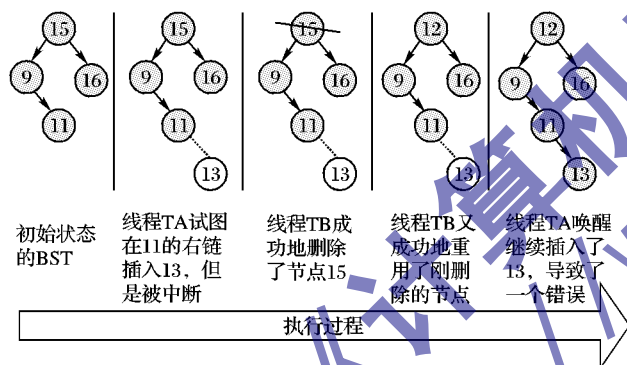


图1 一个错误的节点重用情况

为了防止这种错误,当一个逻辑删除节点满足条件3时,当它的左子树上有正在进行的 insert 操作时,要阻止它自己的重用操作。对于一个满足条件3的 insert(k) 操作,在重用它之前它必须对关键节点进行增加 longnumber 的操作。在增加 longnumber 之后,如果线程发现它自己是第一个(longnumber = 1)到达该节点的,它便可以尝试去重用这个关键节点。稍微后到达的线程(这些线程将会发现节点的 longnumber 大于1)就必须放弃重用该节点的尝试而是马上新建一个节点插入到该 BST 某个叶节点中(插入操作在1.3节中讨论)。当一个线程的 insert(k) 完成以后,它必须减少它所占有关键节点(longnumber 值。对 longnumber 的增加和减少操作都是原子操作。下面给出条件3下节点重用成功的步骤:

- 1) 原子地增加关键节点的 longnumber 值;
- 2) 如果关键节点的 longnumber = 1,则转到3);否则放弃本次重用尝试;
- 3) 尝试将关键节点的状态通过 CAS 操作从“DELETED”改成“CHANGING”;
- 4) 验证关键节点的 key 值从它变成关键节点以后到现在正在进行的操作的时刻是否没有改变。若没有改变,则将 k 赋值给关键节点的 key 然后把 status 改成“CLEAN”;如果发

生改变了,则放弃这次重用。

1.3 插入操作 insert(k)

定义2 对于一个 insert(k) 操作,若新建一个 key 值为 k 的新节点可以链在某个节点 n 上,那么这个 n 节点称为这个 insert(k) 操作的插入点(Insert Point, IP)(在不考虑重用节点的情况下)。

insert(k) 操作尝试将键值为 k 的节点插入到 BST 中,它会有两种返回值:TRUE,说明键值 k 成功地插入到了 BST 中;FALSE,说明 insert 操作失败。其中返回 TRUE 的情况分为两种:1)成功重用一个逻辑删除节点;2)成功新建一个节点,然后链接到 BST 中。insert(k) 操作失败的唯一情况应该是 BST 中已经存在 key 值为 k 的节点。一个 insert(k) 操作可以依据 k 当时满足的条件分为5种情况:

Case 1 k 满足条件2,例如在图2中 insert(15);

Case 2 k 满足条件3,而且它的插入点是它关键节点左子树中的最右的节点,例如在图2中 insert(14);

Case 3 k 满足条件3,而且它的插入点是它关键节点右子树中的最左的节点,例如在图2中 insert(16);

Case 4 k 不满足条件1,而且它的插入点是其父节点的左子节点,例如在图2中 insert(18);

Case 5 k 不满足条件1,而且它的插入点是其父节点的右子节点,例如在图2中 insert(22)。

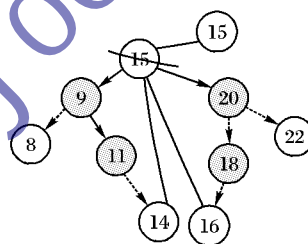


图2 insert 操作的5种情况

insert(k) 操作分为两个阶段:搜寻阶段和寻找插入阶段。如果 k 满足 Case 1,在搜寻阶段就会重用关键节点后返回,否则返回插入点;插入阶段;如果 k 满足 Case 2 或者 Case 3,那么将会尝试去重用关键节点,重用节点成功便返回 TRUE;如果前面重用节点失败或者 k 满足 Case 4 或者 Case 5,那么就创建一个键值为 k 的新节点链到 BST 中。insert(k) 的伪代码如下:

算法1 Insert。

输入 root, k 。

输出 result。

result = FALSE;

criticalNode = insertPoint = parent = NULL;

criticalNodeKey = insertPointKey = 0;

WHILE TRUE

phase1 = searchAndAttemptReuse (criticalNode, insertPoint, criticalNodeKey, insertPointKey);

IF phase1 == 0

// k 满足条件2,重用关键节点成功

result = TRUE;

break;

ELSE IF phase1 == 1

//发现一个有效的节点而且其 key 值等于 k

break;

ELSE IF phase == 2

// k 不满足条件2,或者满足条件2但是重用关键节点失败

phase2 = tryReuseAndInsert (criticalNode, insertPoint,

```

criticalNodeKey, insertPointKey, k);
IF phase2 = TRUE
    result = TRUE;
    break;
ELSE
    //插入失败, 重新从 searchphase 阶段开始
    goto begin;
END IF
END IF
END WHILE

```

1.3.1 搜寻阶段 searchAndAttemptReuse

在搜寻阶段, 从根节点开始寻找插入点, 这个常规的树是一样的。由于加上了节点重用策略, 这个阶段会出现以下情况:

1) insert(k) 操作满足 Case 1 则尝试重用关键节点, 重用成功就直接返回 TRUE; 否则从根节点开始重新搜寻阶段的过程。

2) insert(k) 操作满足 Case 2 或者 Case 3, 那么就要存储它的关键节点、插入点以及插入点的父节点。

3) insert(k) 操作满足 Case 4 或者 Case 5, 那么也要存储它的插入点以及插入点的父节点。

在 Case 2 ~ Case 5 的情况中, 要寻找到可能的关键节点 (如果存在的话)、插入点以及插入点的父节点。下面的方法可以实现这个目的: 把节点看成方向的集合, 左边的方向用 L 表示, 右边用 R 表示, 那么所有的节点都可以看成是从根节点开始的一个方向序列的集合。图 2 中的节点 9 就表示为 L, 节点 17 就表示为 R, 节点 11 就表示为 LR, 以此类推。基于这样的定义, Case 2 ~ Case 5 就可以定义成下面的模式:

1) Case 2: 关键节点的子节点是 L, 接下来的节点都是右子节点, 简写成 LRR+, 也就是 L 后面至少跟着 2 个 R。图 2 中的节点 14 就是 LRR。

2) Case 3: 关键节点的子节点是 R, 接下来的节点都是左子节点, 简写成 RLL+, 也就是 R 后面至少跟着 2 个 L。图 2 中的节点 16 就是 RLL。

3) Case 4: 由于这种情况没有关键节点, 从插入点开始标记, 模式为 LR 或者 LL。

4) Case 5: 同样也没有关键节点, 从插入点开始标记, 模式为 RL 或者 RR。

可以定义一个有限状态自动机来识别这些模式。定义如下: DFA $A = (Q, MYM, D, S_0, F)$; 其中 $Q = \{S_0, S_1, S_2, S_3, S_4, S_5, S_6\}$; 状态的意义如下: S_0 代表初始状态; S_1 代表在节点的方向 LR 字符串的末尾只有 R, 对应着 Case 5 中的 RR 子模式; S_2 代表着方向字符串的末尾是 RL, 对应着 Case 5 中的 RL 子模式; S_3 代表着出现一个 R 后连续出现了 2 个以上 L 结束的情况, 对应着 Case 3 的 RLL 模式; S_4 代表在节点的方向 LR 字符串的末尾只有 L, 这对应着 Case 4 中的 LL 子模式; S_5 代表着方向字符串的末尾是 LR, 对应着 Case 4 的 LR 子模式; S_6 代表着出现一个 L 后连续出现 2 个以上的 R, 对应着 Case 2。D 为转换函数的集合, 定义如下: $D(S_0, R) = S_1$; $D(S_0, L) = S_4$; $D(S_1, R) = S_1$; $D(S_1, L) = S_2$; $D(S_2, R) = S_5$; $D(S_2, L) = S_3$; $D(S_3, R) = S_5$; $D(S_3, L) = S_3$; $D(S_4, R) = S_5$; $D(S_4, L) = S_4$; $D(S_5, R) = S_6$; $D(S_5, L) = S_2$; $D(S_6, R) = S_6$; $D(S_6, L) = S_2$; $dfaRecognizer(curr, instKey, criticalNode, criticalNodeKey, parent, state, n, extTurn)$ 函数用来识别状态的。

searchAndAttemptReuse 的伪代码如下:

算法 2 searchAndAttemptReuse。

输入 root, k 。

输出 criticalNode, parent, insertPoint, oldCriticalNodeKey, oldInsertPointKey。

```

curr = root;
result = 2;
state = S0;
criticalNode = NULL;
WHILE curr != NULL;
    insertPointKey = k;
    IF insertPointKey == k
        IF curr.status == CLEAN
            result = 0;
            break; // "curr" 是一个 key 值为 k 的有效节点
        ELSE
            IF CAS(curr.status, DELETED, CHANGING) = TRUE
                IF cur.key != instKey
                    curr.status = DELETED;
                    curr = root;
                    state = S0;
                    continue;
                ELSE
                    curr.status = CLEAN;
                    result = 1;
                    break;
                END IF
            ELSE
                curr = root;
                state = S0;
                continue;
            END IF
        END IF
    ELSE
        IF insertPointKey < k
            nextTurn = R;
        ELSE
            nextTurn = L;
        END IF
        dfaRecognizer(curr, instKey, criticalNode, criticalNodeKey,
            parent, state, nextTurn);
        IF nextTurn == R
            IF curr -> right != NULL
                parent = curr;
            END IF
            curr = curr -> right;
        ELSE
            IF curr -> left != NULL
                parent = curr;
            END IF
            curr = curr -> left;
        END IF
    END IF
END WHILE
RETURN result;

```

1.3.2 插入阶段 tryReuseAndInsert

在插入阶段, 如果插入操作满足 Case 2 或者 Case 3, 将会尝试重用关键节点, 如果满足 Case 4 或者 Case 5, 那么插入操作则新建一个 key 值为 k 的节点链接到 BST 中。

tryReuseAndInsert 的伪代码如下:

算法 3 tryReuseAndInsert。

```

    输入 criticalNode, parent, insertPoint, criticalNodeKey,
    insertPointKey。
    输出 result。
    result = FALSE
    IF criticalNode! = NULL
        rc = atomic_increase( criticalNode. logNumber );
        IF rc == 1 //满足 Case2 或者 Case3 的条件
            IF CAS( criticalNode. status, DELETED, CHANGING ) == TRUE
                IF criticalNode. key == criticalNodeKey
                    criticalNode. key = k;
                    criticalNode. status = CLEAN
                    result = TRUE;
                ELSE
                    criticalNode. status = DELETED
                    IF attachWithCriticalNode( criticalNode, parent, insertPoint,
                        criticalNodeKey, insertPointKey, k ) == TRUE
                        result = TRUE;
                    END IF
                END IF
            ELSE
                IF attachWithCriticalNode( criticalNode, parent, insertPoint,
                    criticalNodeKey, insertPointKey, k ) == TRUE
                    result = TRUE;
                END IF
            END IF
        ELSE IF attachWithCriticalNode( criticalNode, parent, insertPoint,
            criticalNodeKey, insertPointKey, k ) == TRUE
            result = TRUE;
        END IF
        atomic_decrease( criticalNode. logNumber );
    ELSE
        IF attachWithCriticalNode( criticalNode, parent, insertPoint,
            criticalNodeKey, insertPointKey, k ) == TRUE
            result = TRUE;
        END IF
    END IF
    RETURN result;

```

attachWithCriticalNode 的伪代码如下:

算法 4 attachWithCriticalNode。

```

    输入 criticalNode, insertPoint, parent, criticalNodeKey,
    insertPointKey。
    输出 result。
    1) result = FALSE;
    2) atomic_increase( insertPoint. logNumber );
    3) IF insertPoint. key == insertPointkey
    4) WHILE criticalNode. status == CHANGING
    5) do nothing
    6) END WHILE
    7) IF insertPointKey < criticalNode. key
    8) IF k < criticalNode. key and k > insertPointKey
    9) IF CAS( insertPoint. right, NULL, getNode( k ) ) ==
        TRUE
    10) result = TRUE;
    11) END IF
    12) ELSE IF k < insertPointKey
    13) IF CAS( insertPoint. left, NULL, getNode( k ) ) ==
        TRUE
    14) result = TRUE;
    15) END IF
    16) END IF

```

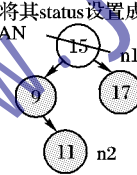
```

    17) ELSE IF insertPointKey > criticalNode. key
    18) IF k > criticalNode. key and k < insertPointKey
    19) IF CAS ( insertPoint. left, NULL, getNode
        ( k ) ) == TRUE
    20) result = TRUE;
    21) END IF
    22) ELSE IF k > insertPointKey
    23) IF CAS ( insertPoint. right, NULL, getNode
        ( k ) ) == TRUE
    24) result = true;
    25) END IF
    26) END IF
    27) END IF
    28) END IF
    29) atomic_decrease( insertPoint. logNumber );
    30) RETURN result;

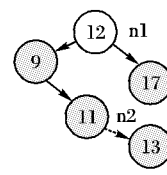
```

对于满足 Case 2 或者 Case 3 条件下的重用操作时必须很小心处理 longnumber 域,因为此时可能也有其他线程的插入操作在也正在重用同一个关键节点。只有第一个到达的线程才会有机会去尝试重用,其他的到达线程发现已经有线程比自己先到达,它们就调用 attachWithCriticalNode() 新建一个节点链接到 BST 中。而且后面到达的线程应该要等待第一个到达的线程完成了操作,节点状态变成“CLEAN”才能调用 attachWithCriticalNode() 方法,图 3 说明了为什么需要这种额外的处理。

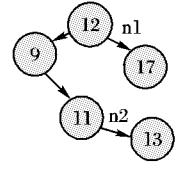
TA 改变节点 n1 的 key, 而且将其 status 设置成 CLEAN



(a) 两个线程 TA、TB 在同一时刻分别尝试插入 key 值 12、13 到 BST 中



(b) 当 TA 正在重用节点 n1 的某个中间时刻, TB 尝试新建一个节点链接到 BST 中



(c) TA 和 TB 都完成了各自的操作, 却导致了错误的结果

图 3 错误的插入情况 1

在 attachWithCriticalNode 的实现中,当关键节点的状态变成 CLEAN 时,后到达的线程仍然需要验证它是否可以插入新的节点(算法 attachWithCriticalNode 中第 4)~6)行,7)~9)行,13)行和 18)~19)行),如果可以的话,就新建一个节点通过一个 CAS 操作链接到 BST 中。程序中的第 2)行中也原子地增加了插入点的 longnumber 域在试图链接新节点之前,这也是为了保护避免被其他的操作干扰。考虑这样一个情况:有一对线程,TA 和 TB,它们都是插入操作,而且关键节点也是同一个节点(如图 4(a)),TA 先到达关键节点,然后 Ta 将尝试去重用关键节点 n1,TA 完成以后 TB 将新建一个 key 为 17 的节点链接到插入点 n3 上,此时又有一个线程 TC 开始运行,它有一个连续的 insert(15)后 remove(16)和 insert(17)操作序列(图 4(b)~(e)),然后线程 TC 成功地将一个新节点链接到 BST 中(图 4(f)),但是这又导致了一个错误的 BST 状态。

当有多于 1 的线程都尝试在同一个插入点插入新的节点时,失败的线程就要重新从搜寻阶段开始新的插入操作。对于新建节点然后链接到插入点中还存在另外一种没有关键点的情况,这种情况用 attachWithoutCriticalNode() 方法处理,

它和 attachWithCriticalNode 类似,不同的地方是还要增减插入

点的父节点的 longnumber 域来保证正确性(见图 5)。

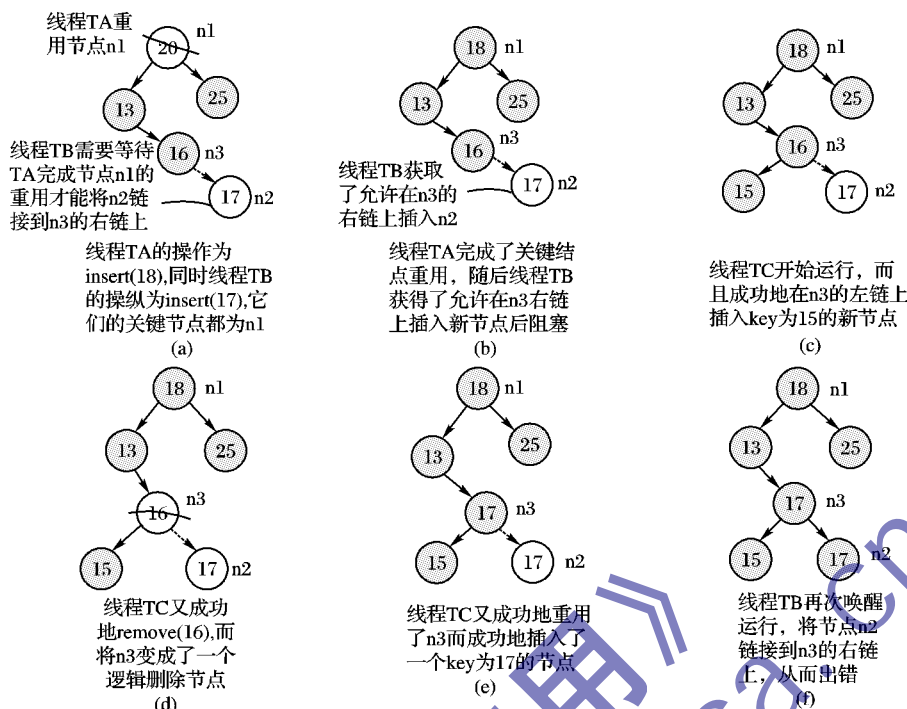


图4 错误的插入情况 2

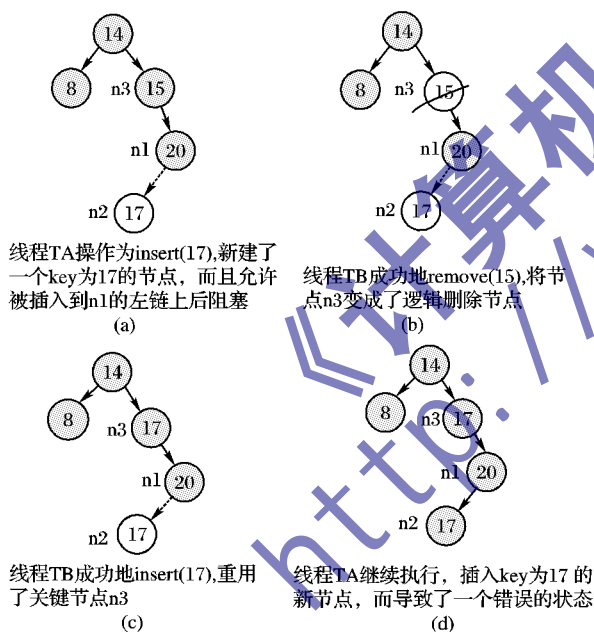


图5 错误的插入情况 3

1.4 查找与删除 find(k), remove(k)

find(k) 从根节点 root 开始遍历寻找 key 值匹配的节点, BST 中存在有效的 key 等于 k 的节点, 函数返回该节点, 否则函数返回 NULL。这个操作和串行化的树是类似的, 只是多了要检查节点的状态来确定节点是否是有效的。

remove(k) 首先要调用 find(k), find(k) 的返回值将会影响到 remove(k) 的返回值: 如果 find(k) 的返回值是 NULL, 那么树中不存在这个节点, 那么 remove(k) 将返回 FALSE; 如果 find(k) 的返回值不为 NULL, remove(k) 则将尝试原子的将节点的状态从 CLEAN 改成 CHANGING, 不成功的话返回 FALSE, 如果成功的话, 然后验证该节点 key 是否仍然为 k 。如是, 则将 status 从 CHANGING 改成 DELETED

并且返回 TRUE; 否则返回 FALSE。

2 实验结果

本文采用目前性能最好之一的 Bronson 的 SnapTreeMap^[6] 作对比分析, 它提供的是 Java 实现的版本, 本文的 BST 使用 C++ 实现。虽然实现语言不同, 但是性能可以对比的: 根据文献[9-12], Java 代码在优化以后性能上要比 C++ 低 2.5~3 倍的速率。因此对比时将 SnapTreeMap 得到的结果放大了 3 倍。

本文使用的是主频为双核 2.66 GHz 的 4 硬件线程的 Intel i5-520 处理器, 操作系统是 Windows 7 专业版。评估性能的方法是测试每毫秒下的吞吐量。随机进行插入、删除和查找操作, 还将这三种操作次数的比例模仿成不同的比例, 而且线程的数量和 key 值的范围都进行了变化, 每个线程都执行 1×10^6 次操作(包含 insert、remove 和 find 操作), 实验中测试的主要数据如下:

1) 吞吐量, 也就是每毫秒的操作数统计, 与 Maurice Herlihy^[13] 用到的方法相同, 实验结果见图 6。

2) 本文的 BST 中逻辑删除节点的比例, 结果见表 1。实验结果表明本文的 BST 在性能上有着非常好的表现, 大致有着 2~8 倍的性能提升, 而且在可扩展性上也是表现突出, 在线程数量达到 128 时仍然有着上升的趋势。性能上升主要有两个方面的原因: ① 当一个 insert 操作可以通过节点重用完成的时候只需要两个 CAS 原子操作, 代价明显降低; ② remove 操作是 wait-free 的。图中数据表明不同的键值范围和不同的操作比例对算法的性能影响也是显著的。当键值范围增大时, 吞吐量呈现下降趋势。这是因为键值增大时, 树的深度也会增大导致搜索路径变长, find 操作中的时间增加, 由于 insert 和 remove 操作也都包含 find 操作, 从而导致吞吐量下降; 另外由于本文是产生键值范围内的随机键值进行操作, 明显当键值增大时, 随机键值相同的概率就会减少, 而当键值相同的同时 insert 操作可能会直接失败返回或者成功直接重用 key 值相同的节点。由于后者的两种操作都是代价很小的, 因此这个方面也会使得吞吐量有所下降。当 insert 操作比例

增加时,吞吐量也会呈现下降的趋势,这是由于 insert 操作中的 CAS 同步操作较多而导致的,相比而言,remove 操作只有一个 CAS 同步操作。

表 1 中说明当 insert 和 remove 比例不变时,随着键值的和线程数的变化,逻辑删除节点的所占比例并无明显变化,这也

说明逻辑删除节点的比例取决于 insert 和 remove 操作的比例。作者还测试了在高负载下(超过 16 个争用的线程) insert (k) 操作中,当 k 满足 Case1 ~ 3 时,insert 操作能成功重用关键节点的概率高达 99.7%,这也说明重用策略是健壮的。

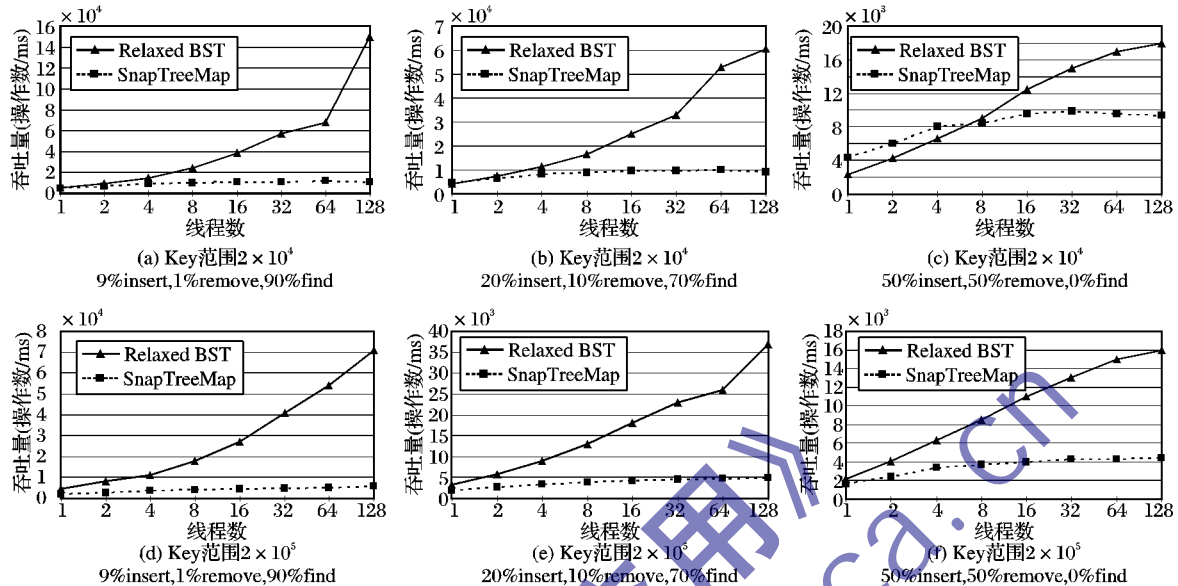


图 6 不同键值范围和不同线程数量下的吞吐量

表 1 不同键值范围不同线程数下逻辑删除节点占总节点数的比例

键值	9% insert, 1% remove, 90% find				20% insert, 10% remove, 70% find				50% insert, 50% remove, 0% find			
	1	4	16	64	1	4	16	64	1	4	16	64
2×10^3	0.102	0.102	0.102	0.102	0.0345	0.0345	0.0345	0.0346	0.501	0.501	0.502	0.502
2×10^4	0.090	0.091	0.091	0.091	0.0329	0.0330	0.0328	0.0328	0.503	0.503	0.504	0.502
2×10^5	0.080	0.079	0.079	0.079	0.0326	0.0326	0.0326	0.0326	0.501	0.502	0.502	0.502
2×10^6	0.079	0.078	0.079	0.079	0.0326	0.0326	0.0326	0.0326	0.502	0.502	0.503	0.503

注:1,4,16,64 为线程数。

3 结语

本文实验数据说明这种改进的节点重用策略是可行的,而且有着良好的性能提升和可扩展性。本文还检查了实验操作完成后 BST 的输出,节点重用的效率以及逻辑删除节点占的比例,后续研究可能集中在 BST 的平衡操作以及组合其他的高效并发数据结构实现更加有效的数据结构。

参考文献:

- [1] GUIBAS L J, SEDGEWICK R. A dichromatic framework for balanced trees[C]// Proceedings of 19th Annual Symposium on Foundations of Computer Science. Piscataway, NJ: IEEE Press, 1978: 8-21.
- [2] NURMI O, SOISALON-SOININEN E. Chromatic binary search trees: A structure for concurrent rebalancing[J]. Acta Informatica, 1996, 33(6): 547-557.
- [3] BOYAR J, FAGERBERG R, LARSEN K S. Amortization results for chromatic search trees, with an application to priority queues[J]. Journal of Computer and System Sciences, 1997, 55(3): 504-521.
- [4] BARNES G. A Method for implementing lock-free shared data structures[C]// SPAA'93: Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures. New York: ACM Press, 1993: 261-270.
- [5] ELEEN F, FATOUROU P, UPPERT E R, et al. Non-blocking binary search trees[C]// Proceedings of the 29th SIGACT-SIGOPS Symposium on Principles of Distributed Computing. New York: ACM Press, 2010: 131-141.
- [6] BRONSON N G, ASPER J C, HAFI H C, et al. A practical concurrent binary search tree[C]// Proceedings of the 15th SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM Press, 2010: 187-197.
- [7] SHAVIT N. Data structures in the multicore age[J]. Communications of the ACM, 2011, 54(3): 76-84.
- [8] HERLIHY M P, WING J M. Linearizability: A correctness condition for concurrent objects[J]. ACM Transactions on Programming Languages and Systems, 1990, 12(3): 463-492.
- [9] PRECHELT L. Technical opinion: Comparing Java vs. C/C++ efficiency differences to interpersonal differences[J]. Communications of the ACM, 1999, 42(10): 109-112.
- [10] HEYDON A, NAJORK M. Performance limitations of the Java core libraries[C]// Proceedings of the ACM 1999 Conference on Java Grande. New York: ACM Press, 1999: 35-41.
- [11] KLEMM R. Practical guidelines for boosting Java server performance[C]// Proceedings of the ACM 1999 Conference on Java Grande. New York: ACM Press, 1999: 25-34.
- [12] BULL J, SMITH L, WESTHEAD M, et al. A methodology for benchmarking Java Grande applications[C]// Proceedings of the ACM 1999 Conference on Java Grande. New York: ACM Press, 1999: 81-88.
- [13] HERLIHY M, LEV Y, LAUCHARCO V, et al. A provably correct scalable concurrent skip list[EB/OL]. [2010-10-10]. <http://www.cs.tau.ac.il/~shanir/nir-pubs-web/Papers/OPODIS2006-BA.pdf>.