A Case for Relativistic Programming

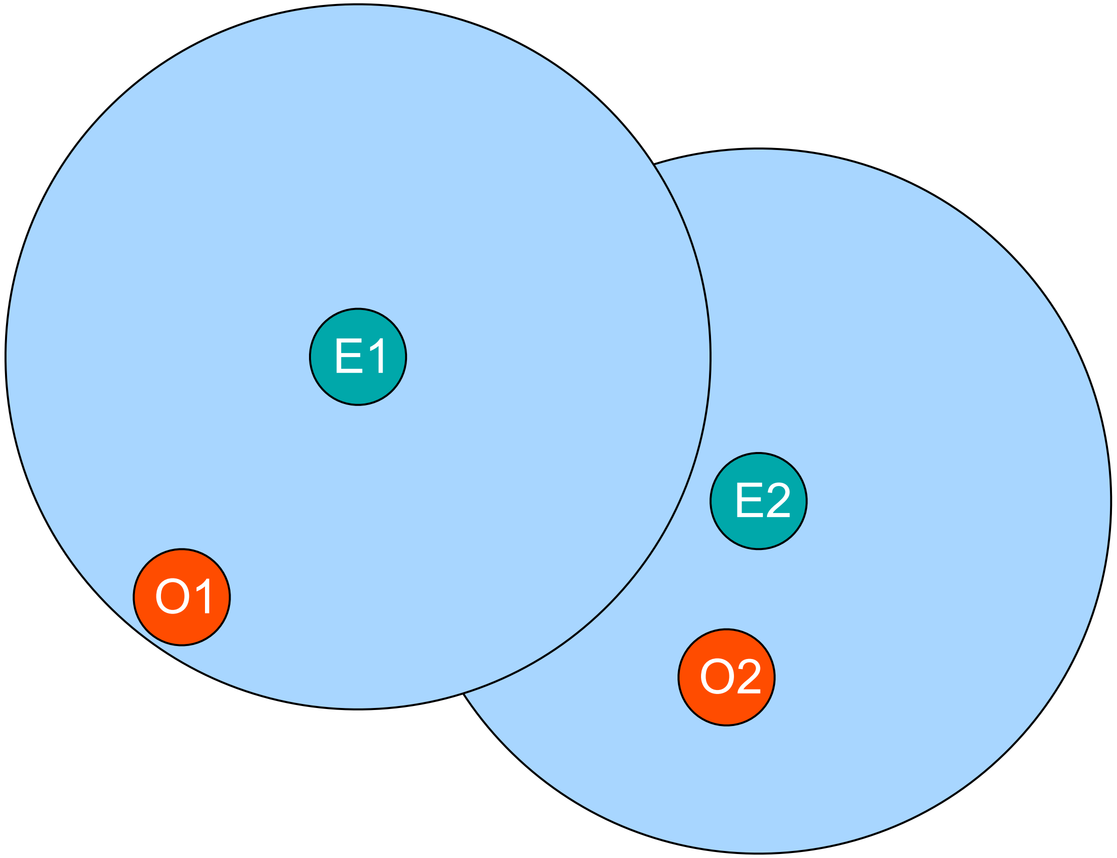
Philip W. Howard

West Virginia University Institute of Technology

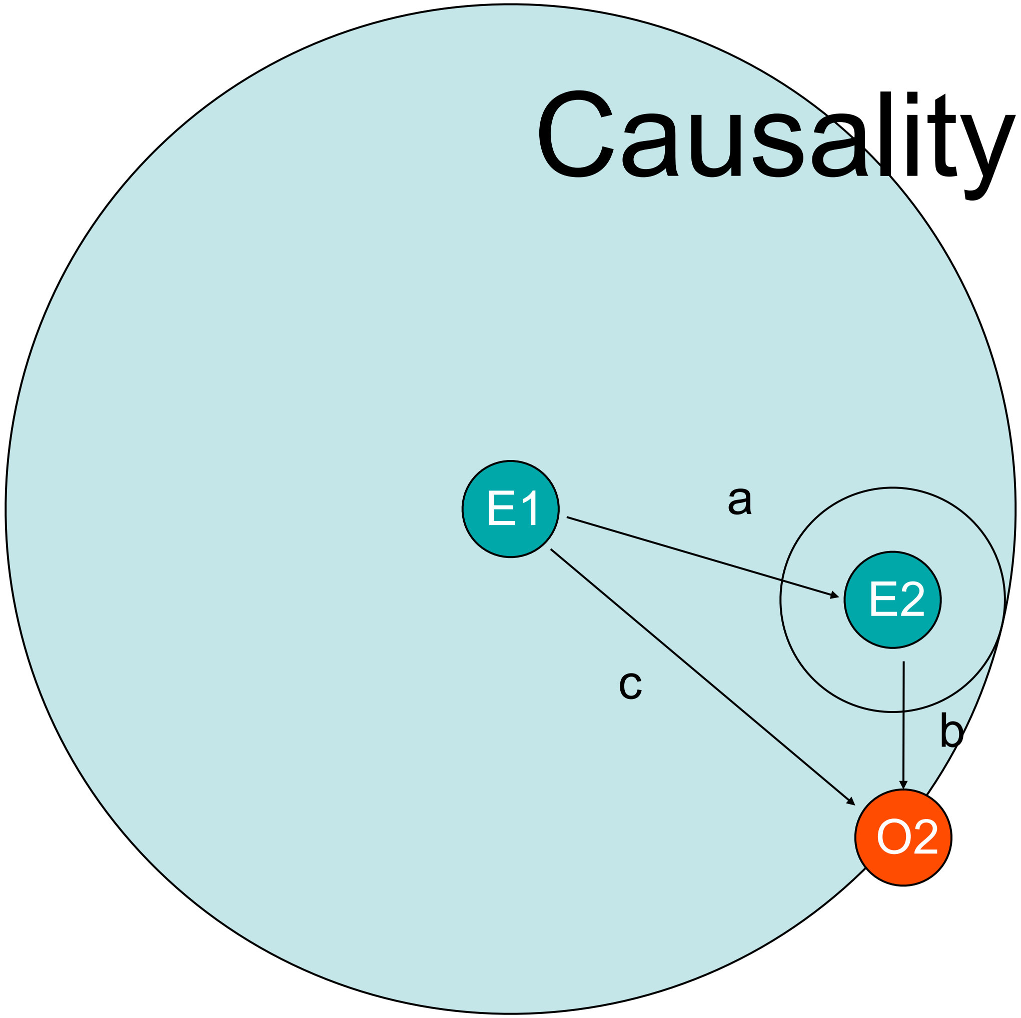
Jonathan Walpole

Portland State University

Events



Causality

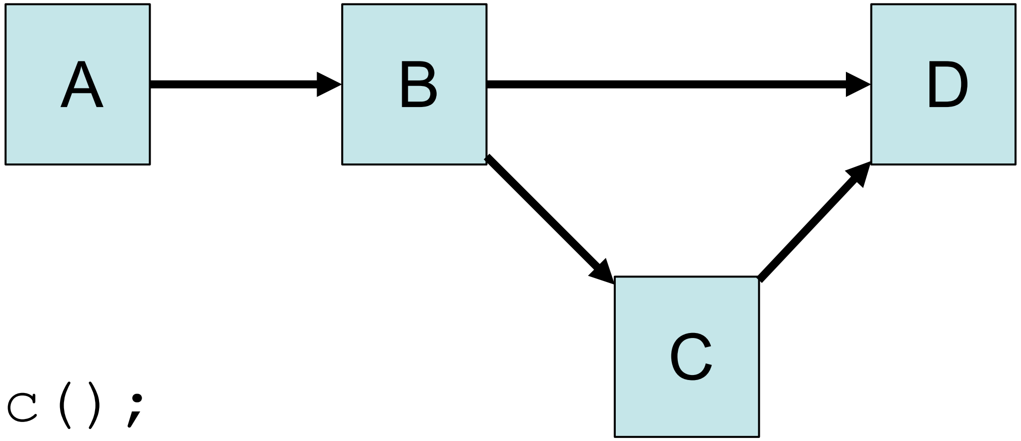


Triangle inequality: $a + b >= c$

Causalty in Computers

- If computers were just wires, we'd be OK
- Computers are wires + illogic

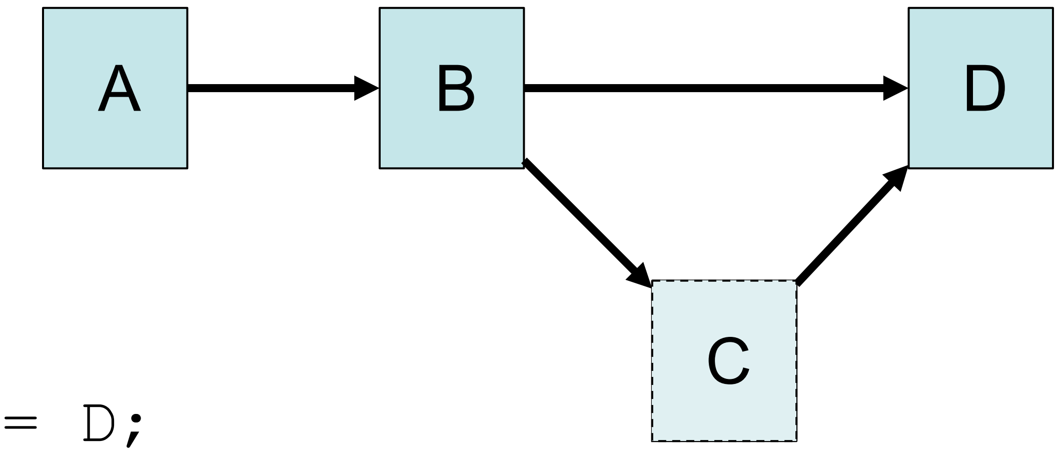
illogic example



```
C = malloc();
C->data = data;
C->next = D;

B->next = C;
```

```
rp-publish(B->next, C);
```

Need for causal delays



```
B->next = D;
wait-for-readers();
free(C);
```

Rules for Placing Relativistic Programming Primitives

1. Write globally visible pointers with `rp_publish()`

2. When performing two writes, one earlier in traversal order and one later in traversal order, separate the two with `wait-for-readers()`

Relativistic vs. Concurrent Balanced Trees

- Relativistic Red Black Trees

- Concurrent AVL Trees

  A practical concurrent binary search tree.
  PPoPP ’10; Bronson et al

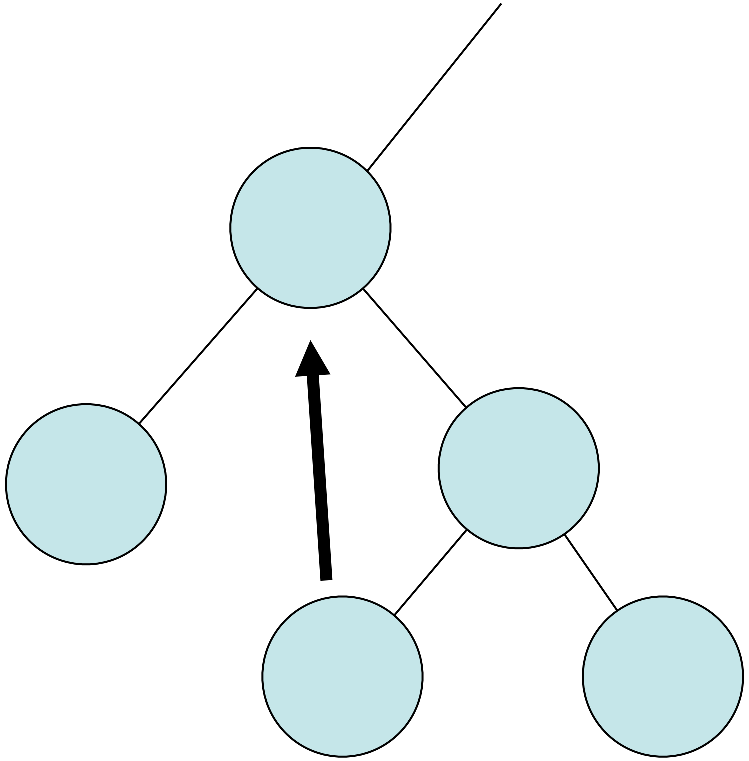
Concurrent AVL get

```
V get(K k) {
    return (V)attemptGet(k, rootHolder, 1, 0);
}

Object attemptGet(
       K k, Node node, int dir, long nodeV) {
    while (true) {
        Node child = node.child(dir);
        if (((node.version^nodeV) & IgnoreGrow) != 0)
            return Retry;
        if (child == null)
            return null;
        int nextD = k.compareTo(child.key);
        if (nextD == 0)
            return child.value;
        long chV = child.version;
        if ((chV & Shrinking) != 0) {
            waitUntilNotChanging(child);
        } else if (chV != Unlinked &&
            child == node.child(dir)) {
            if (((node.version^nodeV) & IgnoreGrow) != 0)
                return Retry;
            Object p = attemptGet(k, child, nextD, chV);
            if (p != Retry)
```

Relativistic RBTree Get

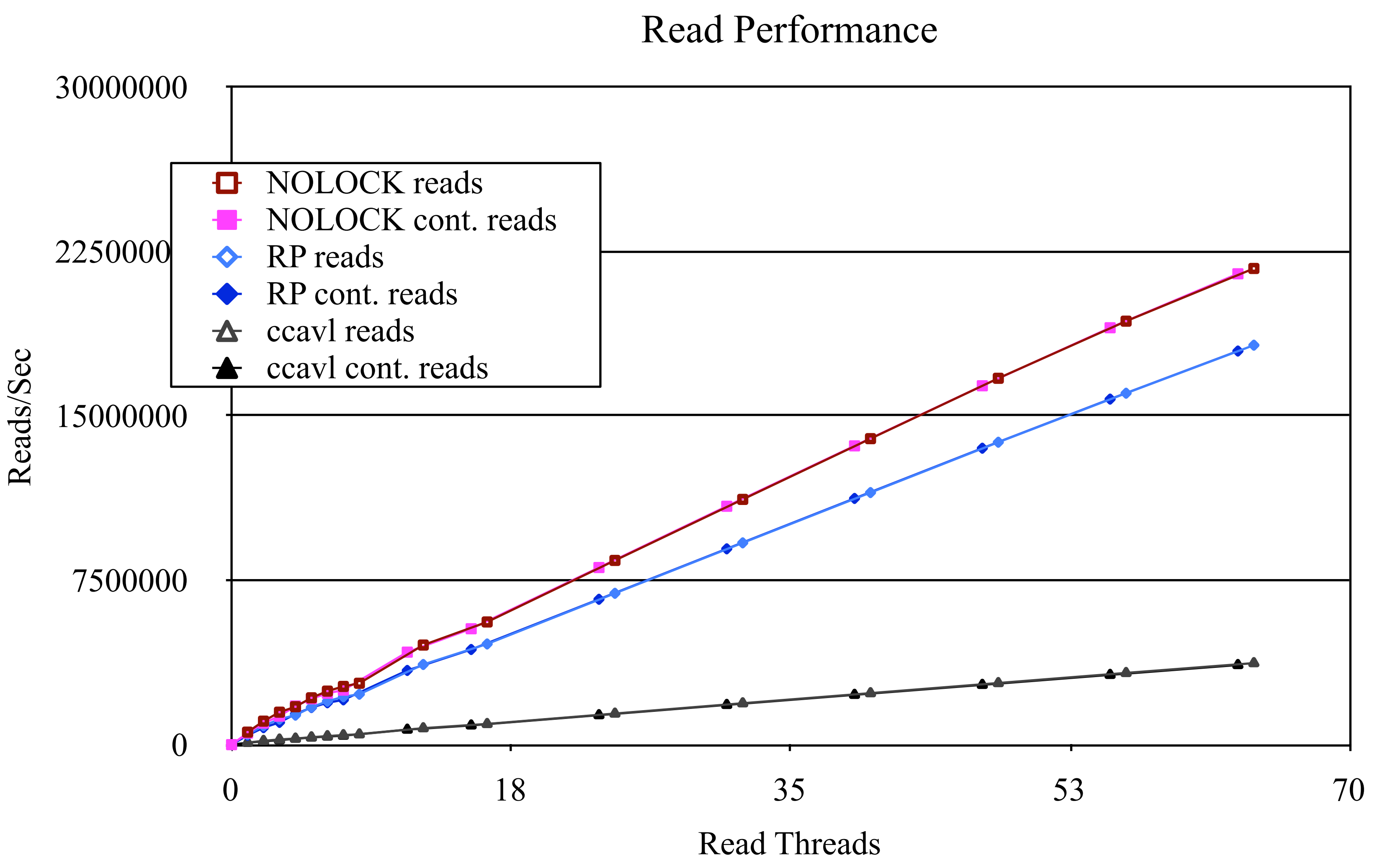
```
void *rb_find(rbtree *tree, long key)
{
    void *value;
    rbnode_t *node = tree->root;


    start_read();


    while (node != NULL) {
        if (key == node->key)
            break;
        else if (key < node->key)
            node = rp-dereference(node->left);
        else
            node = rp-dereference(node->right);
    }


    if (node != NULL)
        value = node->value;
    else
        value = NULL;
```

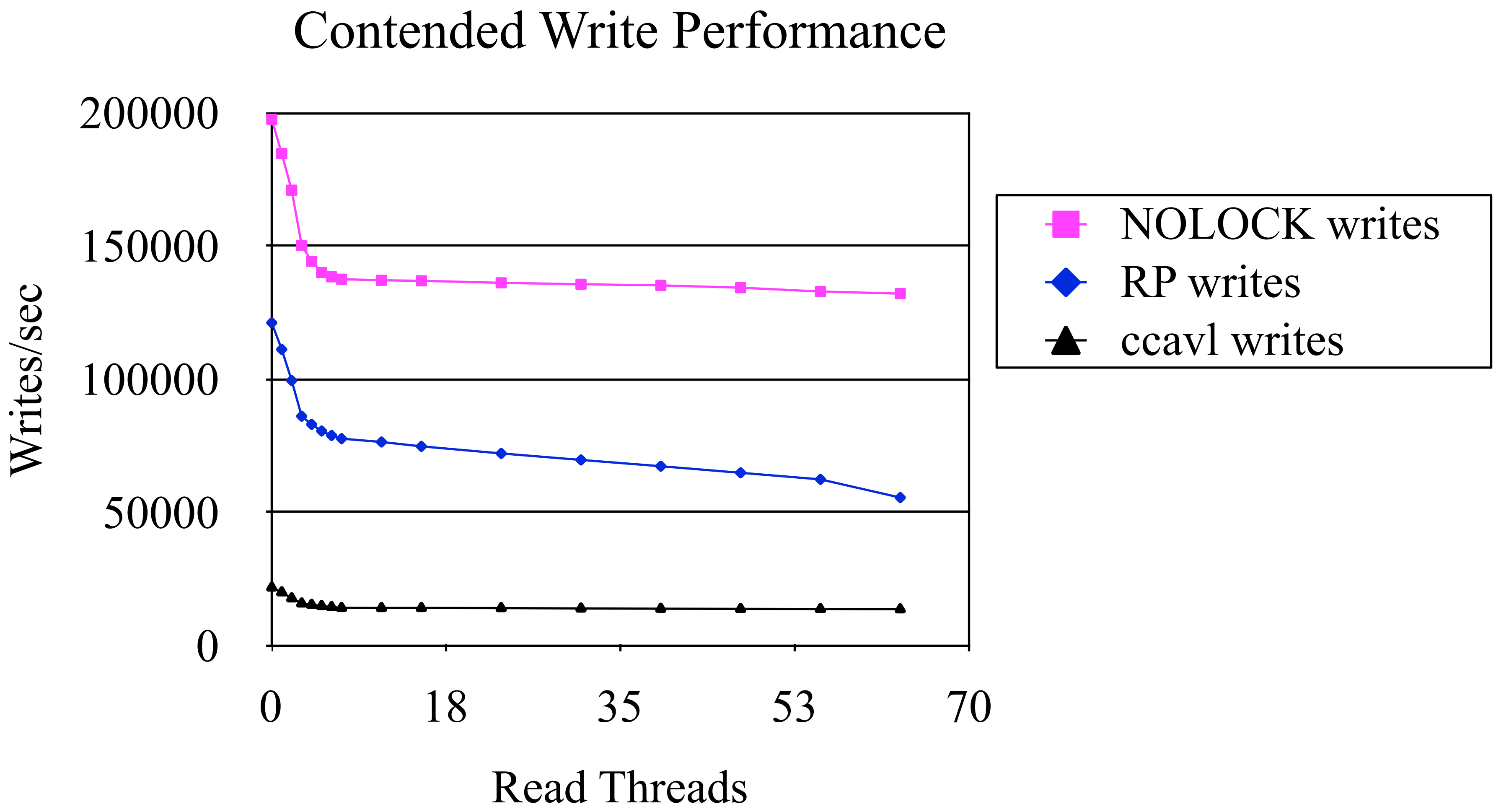
RBTree Delete



RP Delete

```
rbnode_t *new_node = rbnode_copy(swap);

rp-publish(new_node->left, node->left);
node->left->parent = new_node;

rp-publish(new_node->right, node->right);
node->right->parent = new_node;


if (is_left(node))
    rp-publish(prev->left, new_node);
else
    rp-publish (prev->right, new_node);
new_node->parent = prev;

// need to make sure new_node is seen before path to b is erased
wait-for-readers(tree->lock);

prev = swap->parent;
next = swap->right;

rp-publish(prev->left, swap->right);
if (swap->right != NULL) swap->right->parent = prev;
```


Read Performance
30000000
2250000
15000000
7500000
0
Reads/Sec
0
18
35
53
70
Read Threads
NOLOCK reads
NOLOCK cont. reads
RP reads
RP cont. reads
ccavl reads
ccavl cont. reads


Contended Write Performance
200000
150000
100000
50000
0
Writes/sec
0
18
35
53
70
Read Threads
■ NOLOCK writes
◆ RP writes
▲ ccavl writes

Benefits of RP

- High performance, Highly scalable reads
- Simple Code
- Strong correctness properties