javaf / k-compare-single-swap

k-compare single-swap (KCSS) is an extension of CAS that enables atomically checking multiple addresses before making an update.

repl.it/@wolfram77/k-compare-single-swap#readme.md

MIT License

0 stars 0 forks

Star Unwatch

Code Issues Pull requests Actions Projects Wiki Security

master

wolfram77 output in README … 28 seconds ago 14

View code

README.md

k-compare single-swap (KCSS) is an extension of CAS that enables atomically checking multiple addresses before making an update.

AtomicLong63Array provides an array capable of storing 63-bit long values, and supports basic accessor functions like `get()`, `set()` and atomic operations like `compareAndSet()` with either one comparision or k-comparisions. The main purpose of this class is to demonstrate k-compare single-swap (KCSS).

```
get(i):
Gets value at index i.
1. Read value at i (reset if needed).


set(i, v):
Sets value at index i.
1. Convert target value to marked value.
2. Store into internal array.
```

```
compareAndSet(i, e, y):
Performs compare-and-set at index i.
1. Convert expected value to marked value.
2. Convert target value to marked value.
3. Perform CAS.


compareAndSet(i[], e[], y):
Performs k-compare-and-set at indices i.
1. Convert expected values to marked values.
2. Convert target value to marked value.
3. Perform KCSS.


-kcss(i[], e[], y):
Performs k-compare-single-swap at indices i.
1. Load linked value at i0.
2. Snapshot values at i1-rest.
3. Check if captured values match expected.
3a. If a value didnt match, restore (fail).
3b. Otherwise, store conditional new value.
3c. Retry if that failed.


-snapshot(i[], i0, i1, V[]):
Collects snapshot at indices i.
1. Collect old tags at i.
2. Collect old values at i.
3. Collect new values at i.
4. Collect new tags at i.
5. Check if both tags and values match.
5a. If they match, return values.
5b. Otherwise, retry.


-collectTags(i[], i0, i1, T[]):
Reads tags at indices i.
1. For each index, read its tag.


-collectValues(i[], i0, i1, V[]:
Reads values at indices is.
1. For each index, read its value.


-sc(i, y):
Store conditional if item at i is tag.
```

```
1. Try replace tag at i with item.


-ll(i):
Load linked value at i.
1. Increment current tag.
2. Read value at i.
3. Save the value.
4. Try replacing it with tag.
5. Otherwise, retry.


-read(i):
Reads value at i.
1. Get item at i.
2. If its not a tag, return its value.
3. Otherwise, reset it and retry.


-reset(i):
Resets item at i to value.
1. Check if item is a tag.
1a. If so, try replacing with saved value.


-cas(i, e, y):
Simulates CAS operation.
1. Check if expected value is present.
1a. If not present, exit (fail).
1b. Otherwise, update value (success).


-incTag():
Increments this thread's tag.
1. Get current tag id.
2. Increment tag id.


## OUTPUT
Starting 25 threads without KCSS ...
5: done
6: done
20: done
1: done
9: done
8: done
3: done
2: done
```

```
22: done
18: done
16: done
21: done
4: done
13: done
0: done
11: done
12: done
17: done
14: done
10: done
19: done
7: done
24: done
15: done
23: done
{19, 25, 24, 24, 24, 24, 24, 24, 24, 24}
Updates were atomic? false

Starting 25 threads with KCSS ...
0: done
1: done
2: done
3: done
5: done
4: done
6: done
7: done
8: done
9: done
10: done
19: done
18: done
17: done
16: done
15: done
14: done
13: done
12: done
11: done
21: done
20: done
22: done
23: done
24: done
{25, 25, 25, 25, 25, 25, 25, 25, 25, 25}
Updates were atomic? true
```

See AtomicLong63Array.java for code, Main.java for test, and repl.it for output.

references

- The Art of Multiprocessor Programming :: Maurice Herlihy, Nir Shavit

Languages

● **Java** 100.0%

javaf / k-compare-single-swap

Code Issues Pull requests Actions Projects Wiki Security

master

k-compare-single-swap / **AtomicLong63Array.java** / Jump to

wolfram77 word wrapped in egg roll **History**

1 contributor

Raw Blame

275 lines (244 sloc) 6.95 KB

```java
import java.util.*;

// k-compare single-swap (KCSS) is an extension of
// CAS that enables atomically checking multiple
// addresses before making an update.
//
// AtomicLong63Array provides an array capable of
// storing 63-bit long values, and supports basic
// accessor functions like get(), set() and atomic
// operations like compareAndSet() with either one
// comparision or k-comparisions. The main purpose
// of this class is to demonstrate k-compare
// single-swap (KCSS).
class AtomicLong63Array implements Iterable<Long> {
  static final int MAX_THREAD_ID = 1024;
  long[] data;
  long[] tag;
  long[] save;
  // data: contains marked values / tags
  // tag: current tag of respective thread
  // save: current saved value of respective thread

  public AtomicLong63Array(int length) {
    int T = MAX_THREAD_ID;
    data = new long[length];
    tag = new long[T];
    save = new long[T];
  }

  // Gets value at index i.
  // 1. Read value at i (reset if needed).
  public long get(int i) {
    return read(i);
  }

  // Sets value at index i.
  // 1. Convert target value to marked value.
  // 2. Store into internal array.
  public void set(int i, long v) {
    data[i] = newValue(v);
  }

  // Performs compare-and-set at index i.
  // 1. Convert expected value to marked value.
  // 2. Convert target value to marked value.
  // 3. Perform CAS.
  public boolean compareAndSet
    (int i, long e, long y) {
    return cas(i, newValue(e), newValue(y)); // 1, 2, 3
  }

  // Performs k-compare-and-set at indices i.
  // 1. Convert expected values to marked values.
  // 2. Convert target value to marked value.
  // 3. Perform KCSS.
  public boolean compareAndSet
    (int[] i, long[] e, long y) {
    int I = i.length;         // 1
    long[] x = new long[I];   // 1
    for (int o=0; o<I; o++)   // 1
      x[o] = newValue(e[o]); // 1
    return kcss(i, x, newValue(y)); // 2, 3
  }


  // Performs k-compare-single-swap at indices i.
  // 1. Load linked value at i0.
  // 2. Snapshot values at i1-rest.
  // 3. Check if captured values match expected.
  // 3a. If a value didnt match, restore (fail).
  // 3b. Otherwise, store conditional new value.
  // 3c. Retry if that failed.
  private boolean kcss(int[] i, long[] e, long y) {
    int I = i.length;
    long[] x = new long[I];
    while (true) {
      x[0] = ll(i[0]);      // 1
      snapshot(i, 1, I, x); // 2
      if (Arrays.compare(x, e) != 0) { // 3
        sc(i[0], x[0]); // 3a
        return false;   // 3a
      }
      if (sc(i[0], y)) return true; // 3b
    } // 3c
  }

  // Collects snapshot at indices i.
  // 1. Collect old tags at i.
  // 2. Collect old values at i.
  // 3. Collect new values at i.
  // 4. Collect new tags at i.
  // 5. Check if both tags and values match.
  // 5a. If they match, return values.
  // 5b. Otherwise, retry.
  private void snapshot
    (int[] i, int i0, int i1, long[] V) {
    int I = i.length;
    long[] va = V;
    long[] vb = new long[I];
    long[] ta = new long[I];
    long[] tb = new long[I];
    do {
      collectTags(i, i0, i1, ta);
      collectValues(i, i0, i1, va);
      collectValues(i, i0, i1, vb);
      collectTags(i, i0, i1, tb);
    } while(
      Arrays.compare(ta, i0, i1, tb, i0, i1) != 0 ||
      Arrays.compare(va, i0, i1, vb, i0, i1) != 0
    );
  }

  // Reads tags at indices i.
  // 1. For each index, read its tag.
  private void collectTags
    (int[] i, int i0, int i1, long[] T) {
    for (int o=i0; o<i1; o++) // 1
      T[o] = data[i[o]];    // 1
  }

  // Reads values at indices is.
  // 1. For each index, read its value.
  private void collectValues
```

```
    (int[] i, int i0, int i1, long[] V) {
    for (int o=i0; o<i1; o++) // 1
      V[o] = read(i[o]);    // 1
  }

  // Store conditional if item at i is tag.
  // 1. Try replace tag at i with item.
  private boolean sc(int i, long y) {
    return cas(i, tag[th()], y); // 1
  }

  // Load linked value at i.
  // 1. Increment current tag.
  // 2. Read value at i.
  // 3. Save the value.
  // 4. Try replacing it with tag.
  // 5. Otherwise, retry.
  private long ll(int i) {
    while (true) {
      incTag();
      long x = read(i);
      save[th()] = x;
      if (cas(i, x, tag[th()])) return x;
    }
  }

  // Reads value at i.
  // 1. Get item at i.
  // 2. If its not a tag, return its value.
  // 3. Otherwise, reset it and retry.
  private long read(int i) {
    while (true) {
      long x = data[i];
      if (!isTag(x)) return value(x);
      reset(i);
    }
  }

  // Resets item at i to value.
  // 1. Check if item is a tag.
  // 1a. If so, try replacing with saved value.
  private void reset(int i) {
    long x = data[i];
    if (isTag(x))
      cas(i, x, save[threadId(x)]);
  }

  // Simulates CAS operation.
```

```java
  // 1. Check if expected value is present.
  // 1a. If not present, exit (fail).
  // 1b. Otherwise, update value (success).
  private boolean cas(int i, long e, long y) {
    synchronized (data) {
      if (data[i] != e) return false; // 1, 1a
      data[i] = y; // 1b
      return true; // 1b
    }
  }

  // Increments this thread's tag.
  // 1. Get current tag id.
  // 2. Increment tag id.
  private void incTag() {
    long id = tagId(tag[th()]); // 1
    tag[th()] = newTag(id+1);      // 2
  }

  // Gets current thread-id as integer.
  private static int th() {
    return (int) Thread.currentThread().getId();
  }


  // SUPPORT
  // -------
  // Gets length of array.
  public int length() {
    return data.length;
  }

  // Gets iterator to values in array.
  @Override
  public Iterator<Long> iterator() {
    Collection<Long> c = new ArrayList<>();
    synchronized (data) {
      for (int i=0; i<data.length; i++)
        c.add(get(i));
    }
    return c.iterator();
  }

  // Converts array to string.
  @Override
  public String toString() {
    StringBuilder s = new StringBuilder("{");
    for (Long v : this)
```

```java
      s.append(v).append(", ");
    if (s.length()>1) s.setLength(s.length()-2);
    return s.append("}").toString();
  }


  // VALUE
  // -----
  // Creates new value.
  // 1. Clear b63.
  private static long newValue(long v) {
    return (v<<1) >>> 1; // 1
  }

  // Checks if item is a value.
  // 1. Check is b63 is not set.
  private static boolean isValue(long x) {
    return x >= 0L; // 1
  }

  // Gets value from item (value).
  // 1. Copy sign from b62.
  private static long value(long x) {
    return (x<<1) >> 1; // 1
  }


  // TAG
  // ---
  // Creates a new tag.
  // 1. Set b63.
  // 2. Set thread-id from b62-b48.
  // 3. Set tag-id from b47-b0.
  private static long newTag(long id) {
    long th = Thread.currentThread().getId();
    return (1L<<63) | (th<<48) | id; // 1, 2, 3
  }

  // Checks if item is a tag.
  // 1. Check if b63 is set.
  private static boolean isTag(long x) {
    return x < 0L; // 1
  }

  // Gets thread-id from item (tag).
  // 1. Get 15-bits from b62-b48.
  private static int threadId(long x) {
    return (int) ((x>>>48) & 0x7FFFL); // 1
  }

  // Gets tag-id from item (tag).
  // 1. Get 48-bits from b47-b0.
  private static long tagId(long x) {
    return x & 0xFFFFFFFFFFFFL; // 1
  }
}
```

javaf / k-compare-single-swap

Code Issues Pull requests Actions Projects Wiki Security

master

k-compare-single-swap / Main.java / Jump to

wolfram77 output in README History

1 contributor

Raw Blame

95 lines (86 sloc) 2.26 KB

```java
import java.util.*;
import java.util.concurrent.atomic.*;

class Main {
  static AtomicLong63Array shared;
  static int TS = 25;
  static int K = 10;
  // shared: shared array of length K
  // TS: number of threads
  // K: number of comparisions


  // Each thread without KCSS updates all entries
  // of `shared` array to `id+1`.
  static Thread withoutKCSS(int id) {
    long y = id+1;
    return new Thread(() -> {
      try {
      for (int n=0; n<K; n++) {
        shared.set(n, y);
        Thread.sleep(1);
      }
      log(id+": done");
      }
      catch (InterruptedException e) {}
    });
  }

  // Each thread with KCSS updates all entries
  // of `shared` array to `id+1` only if all
  // entries are currently set to `id`
  // (k-comparisions).
  static Thread withKCSS(int id) {
    int[] I = new int[K];
    long[] E = new long[K];
    for (int n=0; n<K; n++) {
      I[n] = n;
      E[n] = id;
    }
    long y = id+1;
    return new Thread(() -> {
      try {
      for (int n=0; n<K; n++) {
        int[] i = Arrays.copyOfRange(I, n, K);
        long[] e = Arrays.copyOfRange(E, n, K);
        while (!shared.compareAndSet(i, e, y))
          Thread.sleep(1);
      }
      log(id+": done");
      }
      catch (InterruptedException e) {}
    });
  }

  // Test with or without KCSS.
  static void testThreads(boolean kcss) {
    shared = new AtomicLong63Array(K);
    Thread[] t = new Thread[TS];
    for (int i=0; i<TS; i++) {
      t[i] = kcss? withKCSS(i) : withoutKCSS(i);
      t[i].start();
    }
    try {
    for (int i=0; i<TS; i++)
      t[i].join();
    }
    catch(InterruptedException e) {}
  }

  // Check if shared data was updated atomically.
  static boolean wasAtomic() {
    for (int n=0; n<K; n++)
      if (shared.get(n) != TS) return false;
    return true;
  }

  // Test both threads without and with KCSS
  // to check if shared data was updated atomically.
  public static void main(String[] args) {
    log("Starting "+TS+" threads without KCSS ...");
    testThreads(false);
    log(""+shared);
    log("Updates were atomic? "+wasAtomic());
    log("");

    log("Starting "+TS+" threads with KCSS ...");
    testThreads(true);
    log(""+shared);
    log("Updates were atomic? "+wasAtomic());
  }

  static void log(String x) {
    System.out.println(x);
  }
}
```