A Symmetric Concurrent B-tree Algorithm

JPF Testing of Java Implementation

Notes on JPF Testing

- Generally, JPF concluded within 3-4 minutes, or else ran out of memory after 3.5 hours.
- If I say "JPF ran too long," it means that it ran for long enough that it didn't seem like it would finish without running out of memory, so instead of waiting 3 hours to see what happened, I stopped it and tried something different.
- In some cases, using fewer threads actually ran longer than more threads.
  - For instance, doing the tests with delete or insert only ran too long with 1 thread, but the same tests on 2 & 3 threads finished quickly.

Code Omissions

- JPF took too long to run on the code used for performance testing.
  - I.e. ran for hours, then ran out of memory.
- Things used only for performance testing were removed
  - E.g. timing, CyclicBarriers
- A while loop that could potentially run forever was turned into a for loop.
- The Critic was removed.
- Size of tree nodes set to 2, so re-balancing happens with very few elements.

Code Modifications Specific to JPF

- Preparation steps, before threads are actually run, were made atomic
  - E.g. creating the threads and pre-inserting some values into the tree before actually running the test threads.
  - Used Verify.beginAtomic(), Verify.endAtomic().
  - Big improvement in JPF run time.
- Verify.getInt() was attempted for randomly-selected values, but took too long even when only a small number of integers, eg 5, were possible choices.
- Assert statements were added (details later)

Used For All Tests

- Listeners:
  - PreciseRaceDetector
- Properties:
  - NotDeadlockedProperty
  - NoUncaughtExceptionsProperty

Configuration Testing

- Errors in the various properties being tested for (false assertions, NoDeadlockProperty, data races) were purposely added to the code.
- Errors added were detected immediately.

Test Method 1: All Thread Types

- Number of insert, search and delete threads are chosen.
- Each search thread is randomly assigned an odd number to search for, which is added to the tree.
    - Search should always return true.
    - Assert statement is used so that JPF will detect an error if a search returns false.
- Each remove thread is assigned a random even number to remove.
- This number is added to the tree, so that the remove actually occurs.
- Each insert thread is given a random even number to insert.

Results

- Various combinations of 4 threads or fewer, including at least one search thread (since search errors are the most important), were used.
- Only one combination of 4 threads finished, after 2.5 hours, without running out of memory.
  - I insert, 2 remove, 1 search: no errors
- 2 threads always finished with no errors.
- About half of the combinations of 3 threads finished with no errors. The others ran out of memory.

Test Method 2: Delete Only

- A number of deleting threads is chosen.
- The threads are each given a random, unique key to delete.
- These keys are added to the tree in advance.
- The delete operation returns a boolean indicating whether the deletion happened or not.
- Assert is used on this value. (Should be true.)

Results

- Errors were found at the assertion statement: The remove operation returned false when it should have returned true.
- The tree, upon examination, *did* have the values removed.
- Error therefore seemed to be in the return value.
- The error did not always occur when more values were added to the tree in advance.

…results continued

- This return value is passed back from a very simple method (where *keys* is a Vector<Integer>, and a lock is already held on the node containing it):

```
public boolean remove_key(int value) {
        if (keys.contains(value)) {
                keys.removeElement(value);
                return true;
        } else {
                return false;
        }
}
```

- Making this method synchronized did not fix the problem.
- This should be possible to debug with ExecTracker.
- I tried and couldn’t figure it out…

Test 3: Insert Only

- A number of insert threads are chosen.
- The threads are each given a random, unique number to insert.
- The insert operation returns a boolean indicating whether the removal happened or not.
- Assert is used on this value. (Should be true.)

Results

- Tests concluded on 2 and 3 threads with no errors.
- Tests ran too long when only 1 thread was used. (Why?)

Conclusions

- Only one error was found in the code.
- The error found is not very serious.
  - It is probably implementation-specific.
  - Might be fixed if I had more time to learn how to interpret the output of ExecTracker.
- Using more than a couple threads at once, and thoroughly testing with Verify.getInt(), took too long and caused JPF to run out of memory. So, there may be undetected errors.
- The code appears fairly reliable.

But…

- This morning, after writing this presentation, I tried using the BFSHeuristic search strategy on a combination of threads (1 of each type) that had run out of memory using other searches.
- It quickly found a bad error: A search for something that was in the tree returned false.
- The code isn’t as great as it seemed. ☹ Though it’s probably good that it took so long to find a serious error.
- Observation: In all the tests I did, either JPF found an error almost immediately, or else it concluded or ran for hours without finding anything.