A. Aiken & K. Olukotun
Winter 14/15

Programming Assignment #2
STM Treap

Due Tue, Feb 3, 11:59PM

In this programming assignment you will use Software Transactional Memory (STM) to implement a concurrent data structure. Unlike the first assignment, the majority of the required work will be spent achieving good performance and scalability, rather than correctness.

Treaps

A treap is a randomized data structure which is simultaneously a heap and a binary search tree. Each node contains both a key and a priority. The keys follow the rules for a search tree: all keys of the left branch of a node are less than the parent node's key, and all keys of the right branch of a node are greater than the parent node's key. The priorities preserve the heap condition: the priorities of a node's children are less than or equal to the priority of the node. If priorities are assigned randomly, the resulting tree is probabilistically balanced. The code for insertion and removal in a treap is simpler than for trees with a strict balance condition, such as AVL or red-black trees.

High-performance concurrent tree structures are difficult because every operation must start traversing the tree at the root. If locks are used, then threads will serialize on the root's lock. Even if reader-writer locks are used, acquiring shared access to the lock may be a source of contention. Reader-writer locks are also tricky because a read lock cannot be directly upgraded to a write lock without a chance of deadlock, but it is not obvious on initial access to a node which mode should be used (this difficulty varies between tree designs).

STM addresses both of the challenges of a concurrent tree. Optimistic concurrency handles simultaneous reads very well, because readers don't need to record their access in a globally-visible location. STM's ability to rollback and re-execute avoids deadlock problems without requiring exclusive write permission to be obtained early.

DeuceSTM

To provide atomicity and isolation for transactions, an implementation of STM must replace direct loads and stores to shared data with read and write barriers. DeuceSTM performs this *instrumentation* by modifying Java class files as they are being loaded into the JVM. This technique is known as bytecode rewriting. To preserve the performance of non-transactional code, each method is split into two versions, one non-transactional and one transactional. Barriers are added only inside the transactional version.

Atomic blocks in DeuceSTM are denoted by attaching the `@org.deuce.Atomic` annotation to a method. If any such method is on the call stack, then a transaction is active. Because DeuceSTM is itself written in Java, it does not attempt to instrument classes from the boot class path. This means that any `java.*` classes are not part of a transaction. If any methods are called on these classes they will execute as if inside an escape action.

What We Give You

IntSet.java – An interface for a set of integers.

Driver.java – A driver for two benchmarks:

Correctness.java – This benchmark that measures the correctness of an IntSet implementation by repeatedly adding and removing elements from the set, and checking against a reference implementation.

Performance.java – This benchmark that measures the performance of an IntSet implementation by counting the number of operations that can be completed in a fixed amount of time. A warm-up period is used to allow time for the JVM's JIT to optimize the code.

CoarseLockTreap.java – A treap implementation that has been made thread-safe by using coarse-grained locking. This will be the starting point for your STMTreap implementation, and also will serve as the baseline for your target performance.

STMTreap.java – Initially just a copy of CoarseLockTreap. Edit this file to complete the assignment.

deuceAgent-1.3.0.jar – The bytecode rewriting agent and STM runtime for DeuceSTM.

pa2.sh – A Torque script for submitting jobs on Amazon EC2.

These resources can be found in /usr/class/cs149/assignments/pa2 on Leland machines such as corn.stanford.edu.

Your Tasks

Edit STMTreap, replacing locks with atomic blocks. Each contains(), add(), or remove() should run in its own transaction. All of your work should be in STMTreap or other new classes. Do not modify IntSet, Driver, Correctness, Performance, or CoarseLockTreap.

Observe the scaling (or lack of scaling) of CoarseLockTreap and STMTreap on a node on Amazon EC2 (see below for instructions). The nodes on Amazon EC2 each an 8-core Intel Xeon E5-2650 running at 2.0 GHz. Good scaling means that up to 8 threads, the ratio of operations-per-second to number of threads is roughly constant.

Modify and/or restructure the code of STMTreap to fix any scaling problems.

Hints

There are two components to parallel speedup: single-thread overhead and scaling. To meet the performance target of this assignment you should focus on the second. Scaling limits in an STM are typically dominated by rollbacks, so to improve the scalability of the STM solution you must minimize conflicts.

As with many aspects of parallel programming, tools for pinpointing the source of performance problems are lacking. DeuceSTM doesn't give feedback about which accesses led to conflict, or even about the ratio of commits to rollbacks. If you just replace the synchronized in STMTreap with @Atomic, every call to add() or remove() will conflict with every other

transaction. To fix this problem you will have to identify the read and write barriers, reason about which conflicts are not actually important to the algorithm, and modify the code to remove those conflicts.

Psuedo-random number generation is a case where atomicity and isolation are often **not** required. DeuceSTM doesn't have direct support for escape actions or open-nested transactions, but you can achieve a similar effect by storing data inside a class that is not instrumented. We suggest storing the randState in a java.util.concurrent.atomic.AtomicLong, and using get() and compareAndSet() to update it.

Once your code is scaling reasonably well, you might try to improve single-thread overheads. Bytecode rewriting requires a much smaller engineering effort than modifying the JIT compiler inside the JVM, but it also limits the optimization opportunities. Some performance gains may be had by copying the results of a transactional read to a local variable, in effect manually performing common sub-expression elimination.

Correctness (40%)

Correctness is the first priority in any concurrent data structure implementation. The Driver class, in correctness mode, tests the add(), remove() and contains() methods of your IntSet implementation and compares the results against a reference, reporting the number of failures (if any). Any failures whatsoever (for any configuration) will result in a **zero** for this section. The script pa2.sh contains the configurations to be used for grading. Please make sure that the pa2.sh script works flawlessly when submitting your assignment.

The pseudo-random number generator need not be protected by the same transactional atomicity and isolation guarantees as the treap. However, your code should generate at most one random value for each state of the generator. Incorrect use of synchronization for the pseudo-random state will result in a 5% deduction (to a minimum of 0%) for this section.

Performance (60%)

Although STM incurs a significant overhead compared to single-threaded performance, your STMTreap should be able to overcome this with sufficient numbers of threads. We will test your program using the Driver in performance mode for three configurations of the benchmark, each of which can contribute up to 20% to the project total. The script pa2.sh contains the configurations to be used for grading. Please make sure that the pa2.sh script works flawlessly when submitting your assignment.

To achieve full points for a particular configuration, your STMTreap must have higher throughput (operations/sec) at some thread count than CoarseLockTreap's throughput with 1 thread. Since the compute node has 8 cores, your best throughput is likely to occur with 8 threads.

If your STMTreap is worse than the baseline at all thread counts, then you will receive partial credit if your STMTreap has a throughput at least 80% of the baseline, at some thread count.

To account for measurement variations and inopportune full GCs, we will run the benchmark up to three times if you don't get full credit, taking the best score. We will use the same Amazon EC2 compute nodes we provide you for these tests.

As an example of a run that receives full credit, consider the following output. The best STMTreap throughput is 1,520,485 operations/sec, which is better than the single-threaded CoarseLockTreap throughput of 1,235,700 operations/sec.

```
Test run for 1000000 range,  5% read...
CoarseLockTreap,  1 threads:  1235700 operations/sec  1235700 operations/sec/thread
CoarseLockTreap,  2 threads:   502793 operations/sec   251397 operations/sec/thread
CoarseLockTreap,  4 threads:   559837 operations/sec   139959 operations/sec/thread
CoarseLockTreap,  8 threads:   533528 operations/sec    66691 operations/sec/thread
CoarseLockTreap, 16 threads:   533215 operations/sec    33326 operations/sec/thread
     STMTreap,  1 threads:   267355 operations/sec   267355 operations/sec/thread
     STMTreap,  2 threads:   442415 operations/sec   221207 operations/sec/thread
     STMTreap,  4 threads:   851281 operations/sec   212820 operations/sec/thread
     STMTreap,  8 threads:  1520485 operations/sec   190061 operations/sec/thread
     STMTreap, 16 threads:  1297700 operations/sec    81106 operations/sec/thread
```

Submission Instructions

You should submit the complete source code for your working solution, as well as a brief text file named README.txt (maximum 1 page) with your name and SUNet ID and an explanation of how the code works and why it is correct. If you would like us to evaluate your performance using DeuceSTM's TL2 algorithm (rather than the default LSA), please modify your pa2.sh appropriately, and make a note in your README.txt.

To submit the contents of the current directory and all subdirectories, log in to a Leland machine such as corn.stanford.edu and run

```
/usr/class/cs149/bin/submit pa2
```

This will copy a snapshot of the current directory into the submission area along with a timestamp. We will take your last submission before the midnight deadline. Please send email to the staff list if you encounter a problem.

Compiling

The pa2.sh script compiles your program and runs both correctness and performance checks under three configurations. While the script is intended to be used with Torque, it is also a valid Bash script and can be run from Linux (or Unix) command line:

```
./pa2.sh
```

Note that DeuceSTM provides two back end algorithms: LSA (the default) and TL2. The back ends may exhibit different performance with your code, and you are free to choose the one which provides you with the best performance. To select TL2, add the following option to the java commands in your pa2.sh script:

```
-Dorg.deuce.transaction.contextClass=org.deuce.transaction.tl2.Context
```

Amazon EC2

While the various Leland and cluster machines are good for development, it is difficult to get accurate performance and scaling measurements on a shared machine. For this purpose CS149 has arranged access to a cluster on Amazon EC2, which consists of a head node, which you can

log in to, and several compute nodes, which feature faster processors with more cores and a substantial amount of memory.

Access to the Amazon compute nodes occurs through a job queue via Torque. The head node that you log in to directly is not intended for running jobs, but controls a number of compute nodes which process the actual jobs. You can use `ssh` to access the head node with a password that we will email to your Stanford email address. If, after we announce the machines are ready, you do not receive an email from us, or if you cannot log in to the Torque head node, please email the staff list to request assistance.

For example,

```
ssh <SUNetID>@<Machine>.compute.amazonaws.com
```

Since Amazon EC2 does not have access to Stanford's AFS, so you must use `scp` or `rsync` to copy your files to and from this system.

A typical workflow for using Amazon EC2 is to edit and compile on Torque head node, then use `qsub` to submit jobs to the queue. All access to compute nodes must occur through `qsub`. Direct `ssh` connections to the compute nodes are not allowed.

Torque

Access to the compute nodes on Amazon EC2 is mediated by Torque. Torque jobs can be either interactive (you get a session directly on the compute node) or non-interactive (you run a script). Either way, you will have to wait in a queue to for access to a compute node. Torque guarantees that when your job runs you have exclusive access to the compute node. Jobs have a maximum time limit of 5 minutes (wall time, not CPU time), and you are only allowed to submit one job at a time.

To submit a job to Torque, use the `qsub` command (from the head node) with the name of a script to run and the directory in which to run it:

```
qsub -d "$PWD" pa2.sh
```

The output from running the script will be stored in a file named `<script>.o.<jobID>` and `<script>.e.<jobID>`. For example, if this was job 15, the standard output would be placed into `pa2.sh.o15` and standard error in `pa2.sh.e15`.

You can also submit interactive jobs, which allow you to obtain a shell on the compute node, similar to logging in to the node directly. To do this, pass `-I` to qsub instead of the name of a script.

```
qsub -I
```

To check the status of jobs in the queue, run `qstat`.

To delete a job listed in `qstat`, either waiting or running:

```
qdel <jobID>
```

Torque is tuned for throughput, not for latency, so it may take longer to schedule your job than you expect, especially when many students are using the system concurrently. We encourage you to begin work early to avoid long wait times immediately before the assignment deadline.