CS 131 Homework 3 Report

Siddharth Joshi

Abstract

This report evaluates concurrency in Java using the various constructs the language provides to deal with the issue of race conditions and the undefined behaviour that may occur as a result. A simple simulation is used to benchmark and compare the performance of synchronized (those that assure sequential consistency) and unsynchronized classes (those that don't) in terms of their runtime and reliability. The principle idea is to analyze the trade offs of inserting sequential checks to achieve greater reliability with the objective of faster runtime. This report also justifies the choices made while implementing a more efficient synchronized class by comparing a few of the packages that provide support for ensuring sequential consistency in Java. The study concludes with a summary comparing the various implemenations of the simulations in terms of being DRF (Data Race Free) and suggests which implementation may be ideal for the corporation GDI.

1 Introduction

Concurrency is a critical part of every modern program and thus is one of the most meaningful criteria with which to evaluate the effectiveness of a programming language. The fundamental challenge in applications relying on concurrency is maintaining correctness or more formally sequential consistency i.e. the concurrency mustn't break the necessary ordering of operations and result in unpredictable behavior. The reason this concern arises is due to the asynchronous nature of the parallel computing where threads can and do execute and differing and unknown speeds, thus it is impossible to predict where another thread may be while one is executing. In general, in keeping with the field's fixation with worst-case analysis, computer scientists generally assume the worst possible interleaving of operations and attempt to guarantee sequential consistency in the case of a such an event as well. The focus of this report is rather different from this notion of an absolute guarantee of correctness. In this report, the aim is to evaluate various degrees of ensuring sequential consistency and analyze the performance tradeoffs this results in.

2 Empirical Testing

2.1 Testing Environment

The UCLA CS Deparment's linux servers were used to run the tests in order to obtain the data. The tests were ran both on *lnxsrv06* and *lnxsrv10*, but for simplicity and brevity only the data from running on *lnxsrv10* has been included as the trend depicted on the two servers was virtually indistinguishable. In particular, the technical specifications of *lnxsrv10* is as following:

Number of Processors: 4

Processor Model Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz

Cache Size / Processor 16896 KB

Core / Processor 4

Total Memory 65799788 kB

Free Memory 1422220 kB

2.2 Runtime v/s Number of Threads

Class	Time per Transition			DRF
	8 Threads	16 Threads	32 Threads	
Null	1427.91 ns	4941.32 ns	6444.41 ns	Yes
Synchronized	3246.38 ns	11227.60 ns	16469.60 ns	Yes
Unsynchronized	5929.34 ns	4100.34 ns	13052.60 ns	No
GetNSet	3486.36 ns	7051.64 ns	16725.30 ns	No
BetterSafe	1909.20 ns	4740.51 ns	9927.85 ns	Yes

The data here shows an increase in time per transition as the number of threads increases and thus indicating that performance suffers as the number of threads increases. This seems contrary to the expected results of multi-threading that is essentially always used as a means of improving performance. However, this particular example doesn't offer many avenues for actual parallelism and hence the overhead of

thread creation along with the performance penalty of either contention between threads (for synchronized classes) or of context switches between threads (for the unsynchronized classes) seem to kill any performance benefit. Therefore in practice this application would not be an application well suited for multi-threading.

2.3 Runtime v/s Number of Swaps

Class	Time per Transition			DRF
	100k swaps	500k swaps	1m swaps	
Null	1336.89 ns	3175.12 ns	1902.53 ns	Yes
Synchronized	2533.99 ns	1555.69 ns	1151.00 ns	Yes
Unsynchronized	1967.06 ns	2112.45 ns	1710.96 ns	No
GetNSet	3255.91 ns	2446.55 ns	1059.50 ns	No
BetterSafe	1768.79 ns	1193.66 ns	632.98 ns	Yes

The data here shows a clear decrease in time per transition and hence increase in performance as the number of swaps increases. This would most probably due to the amortization of the fixed performance penalties of the program that are now split across a larger number of swaps. It is also interesting to note that the performance exhibited by Null seems to dsiagree with the general trend and also contradict our notions regarding its performance (since no work is really done here). This particular data can be considered an anomaly and could have been due to a variety of reasons since the server used is shared across many users.

2.4 Runtime v/s Number of Elements

Class	Time per Transition			DRF
	100 Elements	500 Elements	1000 Elements	
Null	2366.44 ns	1837.88 ns	2796.80 ns	Yes
Synchronized	2725.76 ns	2564.71 ns	2557.62 ns	Yes
Unsynchronized	5387.61 ns	2798.61 ns	3217.17 ns	No
GetNSet	2835.37 ns	2835.42 ns	3119.25 ns	No
BetterSafe	1962.29 ns	1870.51 ns	1920.29 ns	Yes

This data arguably captures the most interesting trend with their being an improvement in performance moving from 100 to 500 elements but a decrease in moving from 500 to 1000. This is likely due to the specifications of the machine that seems to not be able exploit as much spatial locality (perhaps due to cache size) when the size of the array gets extremely large, but does see a performance gain initially due to the amortization of fixed performance costs.

2.5 Summary of Peformance of Classes

From the data it is exceedingly easy to already see that simply from a performance standpoint BetterSafe is superior to the other classes and also guarantees DRF, therefore is probably the best for GDI's application but this is explored more in detail in a section 5.1.

3 Analysis of Packages with Support for Ensuring Sequential Consistency for the BetterSafe Class' Implementation

`java.util.concurrent`

The concurrent package provides a variety of functionality such as classes for sequentially consistent versions of classic data structures like Hash Maps, Skip Lists alongside rather advanced functionality such as *DelayQueue* and *ForkjoinPool* that gives the programmer far more control over the the behavior of the threads - not merely controlling them through simple data locks but actually programming their behavior when they are unable to execute or face some other concurrency challenge. While this functionality would undoubtedly be invaluable to any production level endeavor in the industry, it is definitely overkill for the simple simulation that is being run here and it is unclear if it contains any tools relevant for resolving the task at hand without adding unnecessary complexity. Thus the tools of this would not be well suited to the problem that the BetterSafe class attempts to resolve.

`java.util.concurrent.atomic`

This package as the name suggests provides the programmer with classes implementing basic data structures with atomic accessor and mutator methods. The package contains classes like AtomicBoolean, AtomicInteger and AtomicIntegerArray - the class that is used by the 'partially' synchronous GetNSet class. The reason behind this class being an infeasible solution to ensuring sequential consistency in the program as desired is also why GetNSet is unable to guarantee the desired correctness. In particular, since the class handles all locking for the programmer, it is impossible to actually control the lock and ensure mutually exclusive access for the period of both reading and writing as is required in the simulation (since without this it is possible that the elements being swapped do not violate the check conditions at the time of checking but do so before the write lock is obtained thus leading to incorrect execution). Hence this package too doesn't provide us with a potential solution to the issue of efficient synchronization in the simulation.

`java.lang.invoke.VarHandle`

The class VarHandle within the package java.lang.invoke is a dynamically strongly typed referebce to a variable or to a parametricallly-defined family of variables that provides various modes of access such as: regular read/write, volatile read/write (where the program is forced to actually load and store rather than simply use values kept in registers), and compare-and-swap. In some sense the functionality it provides in the context of the simulation is very similar to that provided by the classes in the atomic package and as such it fails due to the same reason, there doesn't seem to be any

reasonable way of implementing the locking around both read and write using this class.

`java.util.concurrent.locks`

By elimination, it is now obvious that this class must be the one ideal for the implementation of BetterSafe. This is due to the fact that the locks provided by this class allow the programmer the freedom to control the mutual exclusion at much finer granularity and thus ensure that the lock ensures mutual exclusion in the region involving both the read and write, but nothing more. There are various types of locks provided in this package, but after understanding the need of the simulation it is obvious that any benefit that the read-write locks could offer would run into the same issue as the data strucutres provided by the atomic package. Thus, as a result, the lock chosen for the BetterSafe class was the Reentrant Lock that allows the programmer to obtain the lock before carrying out the necessary reads to check if values in the array are valid and the necessary writes to actually swap the values in the desired manner. Moreover, the fine granularity that this lock allows the programmer to operate on, allows for performance gains as evinced by the data presented earlier in this report.

4 Challenges Faced During Performance Testing

The main challenge was generating the data to carry out the tests. It is practically impossible to do this manually hence a script was written to test the programs in the desired manner. This greatly simplified the task of rigorous performance testing and allowed for easy testing with large arrays of random elements alongside differeing number of threads and swaps.

5 Class Analysis

5.1 Best Class for GDI

The best class for GDI would be BetterSafe due to both its superior performance and the fact that is data race free thus doesn't exhibit any unpredictable and/or incorrect behavior. This would not only guarantee the speed desired by customers but also the benefits of absolute correctness as it is Data Race Free.

5.2 BetterSafe v/s Synchronized

The BetterSafe class performs better than the Synchronized class despite both providing essentially the exact same synchronization due to the finer granularity that the BetterSafe class is able to operate on. The *synchronized* keyword locks the entire object when a thread calls the swap method preventing any other thread from interleaving any operation on this class, whereas the finer grained lock ensures this mutual exclusion only in the critical section where the actual reads and writes happen (thus allowing for greater interleaving of operations, thus greater parallelism and resultantly greater performance).

5.3 Data Race Freeness of Classes

The Null Class is DRF trivially as it doesn't actually carry out any operations on the data and hence can never have any data race conditions.

The Synchronized class is obviously DRF due to the use of the word synchronized that ensures that while a thread is executing the method swap no other thread can access or mutate the entire object.

The BetterSafe class again is obviously DRF as it ensures mutual exclusison in the critical section of reads and writes, thus prevents violation of sequential consistency.

The Unsynchronized class is evidently DRF as there is no synchronization method used here and thus threads can interleave their operations arbitrarily allowing the check conditions of ith entry being greater than 0 and jth entry being less than maxval to be violated between a thread's checking and actual execution of the 'swapping'.

The GetNSet class is perhaps most interestingly not DRF as while it does use the atomic data structure AtomicIntegerArray, it is still possible for threads to interleave their write operations in between the read and write of another thread (as while the actual reading and writing is atomic there is no way to guarantee sequential consistency across both actions).