The Report Committee for David Patrick Heidt
Certifies that this is the approved version of the following report:

Framework For Testing Java Concurrency

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Vijay Garg

Herb Krasner

Framework For Testing Java Concurrency

by

David Patrick Heidt, B.A.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2010

Dedication

To Currie, for being supportive.

Acknowledgements

Special thanks to Prof Vijay Garg and Herb Kranser.

11/15/2010

Framework For Testing Java Concurrency

by

David Patrick Heidt, MSE

The University of Texas at Austin, 2010

Supervisor: Vijay Garg

Concurrent programming has become ubiquitous in the arena of application development, requiring most production quality systems to deal with at least some degree of multi-threaded execution. An increasing level of maturity is developing around the impact of concurrency on the design and testing processes. Much of this knowledge focuses on the functional aspect of the design and execution with success measures typically related to the correctness of a program. However, there exists a gap in the research to date around the process for concurrent performance testing. While many companies acknowledge that performance is a major source of complaints in production environments, performance testing historically receives low priority and is often little more than an extension of the functional testing. Possibly the most widely discussed and understood implementation language today, in terms of multi-threaded programming, is Java. The report outlines a standard framework for concurrent performance testing targeted towards Java based applications. In an effort to vet the framework, we execute a series of practical concurrent testing that address some of the most common aspects of concurrent programming in Java, with a particular focus on the Java Concurrency package. As a result, this report presents a portable, extensible framework that designers can use in evaluating the range of concurrency options available in Java within their particular environment. Additionally, it provides specific insight into the performance of these options in a typical run-time environment. This includes particular attention to the comparison of traditional lock based approach to non-blocking algorithms.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1: Introduction

The percentage of software applications that are mutli-threaded has grown and continues to grow in an attempt to take advantage of ever-improving hardware. One of the most popular implementation languages for such applications in the industry today is Java. Consequently, a great amount of collective knowledge has been established withing the development community with regards to concurrent programming in Java. While most of the questions being asked in this area can apply to any programming language, the answers are typically provided in a very Java specific grammar with Java specific implementation details. Much of the research to date has focused on error detection and prediction. This often leaves questions about performance to one-off tests that are commonly extensions of the functional testing. While much existing research focuses on the question "Will a program do what it is supposed to do without error?", this paper focuses on the question " Will a program do what is is supposed to do with sufficient speed?"

This paper continues with a background discussion on the options for handling concurrency in Java. It presents a testing framework geared toward answer the core question posed in this Introduction. This is followed by an execution of a series of typical tests for which the framework would be used. It concludes with a discussion of the contributions provided by the framework and the tests presented.

Chapter 2: Background

Concurrency Options in Java

This section provides a brief summary of the different concurrency models used in Java. Various applications of these models are the target for our proposed testing framework as well as the specific focus area for the tests discussed later. Though these general approaches are independent of the implementation language, we will be focusing on the benefits and drawbacks as they exist in the Java implementation. The three general approaches are intrinsic locking, explicit locking, and non-blocking algorithms.

Intrinsic locking is implemented in Java via the synchronized block. In involves the using the lock associated with each instance of Java Object or a Java Class. Only a single thread can obtain this lock at a time so all actions taken inside of a synchronized block can be considered to be atomic. A typical synchronized block takes the form of the code in Listing 1. It is important to note that the get and the set of the count value in line [Lea05] is not an atomic action by itself. The Java Virtual Machine (JVM) treats this a separate get, increment, and set actions.

```
class IntrinsicLockCounter {

            int count = 0;
            static Object lock = new Object();

             IntrinsicLockCounter() { }

            public incrementCounter() {
                        synchronized ( lock ) {
(1)                                 count = count + 1;
                        }
```

```
          }
}
```

Listing 1: Intrinsic Lock Based Counter

This basic form of synchronization is also the earliest form of synchronization in Java. As such, the conventions associated with intrinsic locking (synchronized keyword and the related notify and wait methods) became well known in the developer community. One of the biggest initial drawbacks of intrinsic locking was the performance. There was a great amount of overhead with maintaining the locks and the threads that were waiting on these locks to release.

In response to the performance issues and the need for developers to repeatedly solve the same concurrency issues, JSR 166 was proposed. The requirements of JSR 166 were implemented in the Java Concurrency package released with Java 5 in 2003. This package provides a standard for the use of explicit locks. This reduced some of the complexity associated with the management of intrinsic locking while providing much greater performance at the time. Listing two shows a different implementation with the same functionality and thread safety as in Listing 1. While this listing provides a solution for a basic multi-threaded problem, the functionality of the locks Java Concurrency package covers a much wider range of concurrency functionality.

```
class ExplicitLockCounter {

        int count = 0;
        private static final Lock lock = new ReentrantLock();

        public ExplicitLockCounter() { }

        public incrementCounter() {
                try {
                        lock.lock();
                        count = count + 1;
                } finally {
                        lock.unlock();
                }
        }
}
```

Listing 2: Explicit Lock Based Counter

The final approach involves the use of non-blocking algorithms. As contention increases in a lock-based scheme, the amount of time the JVM spends managing the waiting threads increased. When the amount of work done while a lock is held is very small - which is often the case since good design suggests minimizing the time a lock is held in order to minimize overall contention - the proportion of overall JVM effort spent on thread management can be excessively high.[Goetz06] Non-blocking algorithms were developed in an attempt to eliminate this overhead. These algorithms are written under the guiding principle that no thread should ever pause execution because of another thread.

Java 5 introduces Atomic variables, a key construct in developing non-blocking algorithms in Java. Atomic variables are primitive wrappers that support a number of basic actions which are executed atomically. For example, AtomicInteger.compareAndSet(int old, int new) will set the value of the of an integer to the new value if current value is the same as the old value. If successful, true is returned. If, for example, another thread has updated the value of the AtomicInteger since the current thread read the value and the old value doesn't match the expected current value, false is returned. At which point, the current thread can try again or continue on depending on the goal of the algorithm. The code listing below shows an example of non-blocking implementation of the counter example.

```
class NonblockingCounter {
        private AtomicInteger value;

        public int incrementCounter() {
                int old, new;
                do {
                        old= value.get();
                        new = old + 1;
                } while (!value.compareAndSet(old, new));
                return new;
        }
}
```

Listing 3: Non-blocking counter

Concurrent Data Structures

In addition to the low level constructs described above, the Java Concurrency package provides several implementations of the common data structure interfaces. Prior to this release, a thread safe data structure was provided by getting synchronized collection from the Collections package which used intrinsic locking. A designer could create his or her own thread safe data structures using intrinsic locking as well. The Java Concurrency package provides implementations of common data structures using the explicit locking constructs introduced at the same time.

Even with the benefit of explicit locks, these implementations still held victim to the general drawbacks of lock-based algorithms. In cases of contention, some threads are required to block on the thread holds the lock. There are also risks of starvation or deadlock depending on the actions taken while a data structure is locked. [Goetz06] For cases where these drawbacks are unacceptable, a non-blocking implementation should be evaluated. The Java Concurrency package provides one such implementation in a non-blocking queue. In making this decision, performance considerations typically weigh heavily. There is not conclusive research to suggest the exact conditions under which one solution should be taken over another.

Existing Performance Testing Research

Pooley highlights a perceived gap in the area of software performance analysis in general. This gap is a lack of any widely accepted and adopted performance analysis techniques. Pooley suggests that this is largely related to failure of software engineers to apply formal analysis techniques to their design, techniques such as Petri nets and Markovian numerical solutions. [Pooley00] In the absence of formally provable testing methods, performance testing is left to more ad hoc methods. Vokolos and Weyuker also discuss the lack research in the area of software performance testing. They point out the irony in the fact that, while the performance of a system is one of the most common subjects of complaints about a deployed system, it is often one of the least tested aspects of the system. [Vokolos98]

Goetz provides a theory as to why performance testing often yields misleading or misleading results. Performance tests are typically derived from functional testing. While this provides value in that there is some assurance that the performance tests are against correctly functioning code, these two activities have distinct goals. [Goetz06]

The topic covered here, concurrent performance testing, is even less explored. Some research has occurred in the area of sequential testing. In this approach, the results are more deterministic since developers explicitly set the path of execution. In concurrent testing, however, repeated testing may result in a wider range of results. [Eytani07] This is part of reason why functional testing does not meet the needs of performance testing.

There has been more research to date in the area of measuring the correctness of concurrent systems. This is more focused on the challenge of identifying design flaws and predicting faults. There is a fair amount of research in the area of synchronization coverage, which aims to validate a testing framework in terms of how well it covers all scenarios related to concurrency. One such effort is ConTest, an internal IBM tool, aimed at providing synchronization coverage [Bron05]. These efforts still focus on the functional aspects of a system, rather than the performance.

Goals of Concurrency Performance Testing Framework

The aim of this project is to provide a framework to support concurrent performance testing. As discussed above, functional and performance testing should be considered two independent activities with independent designs. This framework is only concerned with the performance implications of a design.

There is typically an explicit intent to shy away from the environment and platform specific measurements. [Edelstein03] One goal of this research is to highlight the specific differences. It would be useful to know how strong of a variance there may be across platforms. It is common for the developer community accept general statements about specific behaviors of Java. This led to poor practices like double check locking. Having such a framework would show which idioms are constant and which should be looked at per environment.

In addition to being cross platform, the framework should be able to provide insight across implementations and algorithms. For example, it would be useful to compare blocking versus non-blocking algorithms.

Goals of Performed Tests

The second contribution of this project comes from a set of tests run through the framework. These tests were selected to represent the types of questions the framework should help answer. In general, these tests are comparing various synchronization models under a range of conditions. The concepts and comparisons should be familiar to experienced designers of concurrent systems. In reviewing the tests, one should keep in mind that they a snapshot of relative performance under specific conditions. The goals and parameters of each test are detailed later in the in this report.

Chapter 3: Concurrency Performance Testing Framework

Framework Principles

With the aforementioned goals in mind, four key principles drove the design of the framework. They are as follows:

Flexibility – This refers to the ability to adjust the parameters of the test without additional coding of the test cases. For example, users should be able to easily adjust the number of threads, the timing of the threads, and the behavior of each of the threads with minimal effort.

Usability – This refers to the amount of effort required to get a new test up and running. There should be clear abstraction of the Logic Under Test (LUT) from the framework itself.

Unobtrusive – There should be a minimization of the Observer Effect, where the framework impacts the performance of the Logic Under Test.

Concurrency oriented - There should be specific consideration to the concurrency aspects. For example, there should be clear direction on where synchronization models should be implemented. This should not be a black box performance testing tool.

Framework Design

Emer presents ASim, a modular performance testing framework targeted toward measuring microprocessors. They describe a design that includes separating Feeder modules that provide the actual execution instruction from the Port modules that represent the hardware components that process the actual executions. Their modular design provides a "mix-and-match capability" that allows for the reuse of modules in various contexts. This leads quicker test development and stronger, more mature modules which provide greater confidence. [Emer10]

We adapted this design to fit a more software oriented context. The key modules are Action modules and Actor modules. Actions define the logic that is being synchronized, such as a read or set of a value. The listing below provides the interface for the Action class. The executeAction() method should hold the definition of the Logic Under Test. Designers can associate a delay with an Action that the Actor can take in to account while repeatedly performing the Action.

```
public interface Action {
        public boolean executeAction();
        public int getDelay();
        public void setDelay(int delay);
}
```

Listing 4: Action module

Actors define how these Actions are executed. The key part of this definition are the synchronization management details, which might be lock-based, non-blocking, or even no synchronization. While there are some common predefined Actions and Actors available, these would often be user created modules. The listing below presents the AbstractActor from which all Actors should be derived. There are three abstract methods that should be defined for a given Actor. The act() method contains the core logic that calls the action. It will typically maintain all the synchronization logic for a given test. The init() method should handle any one time initialization required on start-up. The finish() method should manage the shutdown and clean up of the current thread.

```
public abstract class Actor extends Thread {
        protected long actionCount;
        protected final Action action;
        public Actor(Action action) { init(); }
        protected abstract void act();
        protected abstract void init();
        public abstract void finish();
        public long getActionCount() { return actionCount; }
        public void reset() { actionCount = 0; }
        public void run() { act(); }
        public Actor clone() {
                .
                .
        }
        .
        .
}
```

Listing 5: Actor module

This separation of what is done and how it is called allows users to compare different synchronization implementations for the same actions. A test can easily isolate disparities among potential synchronization options for a user. It also allows testers to understand how various combinations of these Actors and Actions interact. While a locking implementation may outperform a non-blocking implementation for one Action, the reverse may be true for another Action.

Once defined, these Action and Actor pairs are executed by the framework engine. The engine handles the various test execution responsibilities, including thread management, test repetition, and metrics collection. The engine is configured through a properties file that provides control of the test parameters. Users can specify the number of executions of test that they would like to allow for averaging of measurements across multiple test runs. They can specific the length of the test run in milliseconds. They can also specify the set of thread counts for which the tests will be run. This directly corresponds to how many Actors are instantiated – Actor extends the Thread object. For example, a user may want to see how a given set Actor/Action pairs perform with ten threads or five threads or even a single thread. The configurable nature of the engine provides another level of isolation for the range of variables that may affect performance in a production environment.

Metrics collection requires a collaborative effort between the Actor modules and the engine. Each Actor is responsible for collection number of successfully executed actions (non-blocking implementations may have many unsuccessful actions). The engine manages the aggregation across threads and tests.

One key aspect of the framework's performance tracking is that data is defined in terms of the number of actions executed. Most existing performance testing frameworks define performance in terms on how long a given action takes on average. While it is certainly possible to calculate this measure by dividing the number of actions by the number of Actors and the length of the test run, the precision of this value will be much lower with this framework. This has to do with how the engine notifies the Actors to shutdown. To minimize the intrusion of the framework, some concessions were made on this immediacy of the shutdown process. This means the specified run time will be honored on a best effort basis, but may run slightly longer. This matches up with the goal of comparing relative behaviors as a part of the design process rather than defining specific operation execution times that may be used as SLAs.

Chapter 4: Test execution

Test definition

This section details the categories of motivating examples for the framework. One category of interest is the cost of intrinsic locking. One aspect of this is evaluating the cost of intrinsic locking across different JVMs. As referenced above, the forces behind Java have advertised marked improvement to the implementation of intrinsic locking since Java 1.3. What is not clear is the scale of these improvements. Another aspect of this is the comparison of intrinsic locking versus locking constructs provided in the Java Concurrency package. This testing category serves to question the general notion that using intrinsic locking is inherently slower that other options and should be avoided where possible. In this category, the LUT should be the same across test run. The variable will be the JVM version on which the tests are run or the locking mechanism.

Another category is the cost of blocking vs non-blocking synchronization algorithms. In the java.util.concurrent package, this will typically translate to the use of ReEntrantLocks vs Atomic variables. Non-blocking algorithms have yet to gain widespread adoption within the development community. This should give data to understand whether or not these algorithms should be considered suitable alternatives from a performance perspective. The LUT should have the same functionality across tests. The variable will be the implementation of the synchronization.

Finally, we are interested in the cost of blocking vs non-blocking data structures. In the Java Concurrency package, this will translate to a comparison of ConcurrentLinkedQueue vs implementations of LinkedBlockingQueue. The LUT should have the same functionality across tests. The variable will be the implementation.

Test environment

All tests were performed in two separate environments with details as follows:

- Processor: AMD Opteron 64-bit, 2.8 GHz
- Memory: 1 GB memory
- O/S: CentOS 5.1 64-bit Linux

The parameters for the tests were as follows:

- **Run time** – All test tests ran from 10sec -30sec. It is commonly agreed that one should keep the length of code being synchronized to a minimum. As such, the brief nature of the Actions under test cause the number of actions performed to quickly approach the maximum value of java primitive types.
- **Number of threads** – All tests were run with 1, 5, and 10 threads. While running in single threaded environment would preclude the need for synchronization, we found it to provide useful additional insight.
- **Number or tests** – All tests were averaged across five test runs. We found that results were very consistent across test runs.

Test results

This section details the tests executed through framework. We translate the high level goals of the tests described above into specific Actors and Actions that fit in to the framework. We detail the behavior of each Action along with the manner in which synchronization is implemented. This is followed by the results and an analysis of the results.

Simple Loop with Intrinsic Locking, Java 1.4 vs Java 6

Action – Runs through a for-loop 1,000,000 times.

Actors – This test contained a single SynchronizedActor.

Results – Table 1 shows that the performance of intrinsic locking is dramatically increased by several orders of magnitude.

Analysis – This test confirms the claims of the authors of Java regarding the increased performance of intrinsic locking. It also confirms the widely held beliefs about the cost of synchronization in earlier versions of Java.

	Number of Threads		
	1	5	10
Java 6	467282072	38596276	37925028
Java 1.4	662	648	632

Table 1 : Simple Loop, Different JVMs

Simple Loop, Intrinsic Locking vs Explicit Locking

Action – Runs through a for-loop 1,000,000 times.

Actors – This test compared the same two actors from the ListContains test, LockingActor and SynchronizedActor.

Results – Data Figure 2 shows that the LockingActor performed much better than the SynchronizedActor in a multi-threaded environment while the reverse was true in a single threaded environment. Obviously, if a program executes in a single thread there would be no need for any synchronization. It is possible, however, that the synchronization pattern for a program may be such that a single thread does a vast majority of the work and there is little contention from other threads.

Analysis – We see here that, despite the improvement in the implementation of intrinsic locking, explicit locks still outperform. This test had a relatively inexpensive Action to highlight the differences between the SynchronizedActor and the LockingActor.

ListContains

Action – Search an ArrayList with 1,000,000 elements for an entry that is not there. ArrayList is not synchronized.

Actors – This test compared two actors, LockingActor and SynchronizedActor. LockingActor uses a ReentrantLock to guard execution of the Action. SynchronizedActor uses the synchronized construct where execution of the action is guarded by obtaining a lock on a static Object.

Results – Data Figure 1 shows that the two Actors provide similar performance, with the LockingActor performing slightly better.

Analysis – In this test, the FUT took considerably locking as evidenced by the low number of actions executed. As expected, when the length of the FUT was increased the relative impact of the synchronization details was marginalized. This highlights the benefit of separate the Action from the Actor. By simply changing the action from the previous test, we observe a much different picture of the relative performance of synchronization models.

Counter

Action – This test contains two actions since the Counter logic would change based on the synchronized model. AtomicCounterAction assume the the fetch and increment actions can be done as an atomic action based on synchronization being handle by the Actor. The NonAtomicCounterAction uses the compareAndSet() method on an AtomicInteger which returns true id the set was successful and false otherwise.

Actors – This test compared three actors, LockingActor, SynchronizedActor, and NonBlockingActor. LockingActor and SynchronizedActor manage the synchronization for the AtomicCounterAction. The NonBlockingActor repeatedly calls the NonAtomicCounterAction until it returns true. Only once the NonAtomicCounterAction returns true, does that action count get increment. If the action returns false, this is not seen as actually having successfully execute the action and the action count is not increase.

Results – Data Figure 3 shows that the LockingActor SynchronizedActor provide similar performance while NonBlockingActor strongly outperforms the other two by orders of magnitude.

Analysis – This compares two locking based algorithm with a simple non-blocking one. We see the explicit locking mildly outperforms the intrinsic locking as in the test above. The non-blocking option, however, show an improvement by orders of magnitude. In this case, it is clear that the non-blocking approach is the preferred choice.

Queue

Action – This test contains two actions BlockingQueueTakeOfferAction and NonBlockingQueueTakeOfferAction. Both actions execute the same steps of taking an item off the queue and placing it back on the end of the queue. They vary by the implementation of the queue. BlockingQueueTakeOfferAction uses a LinkedBlockingQueue that manages synchronization via locking. NonBlockingQueueTakeOfferAction uses a ConcurrentLinkedQueue that manages synchronization using a non-blocking algorithm. Both implementations reside in the java.util.concurrent package.

Actors – This test uses a simple UnSynchronizedActor since the synchronization is managed by the queues themselves. It assumes every action is a success. This is a safe assumption in this test since neither implementation is fail-fast and, consequently, nor will they throw a ConncurrentModificationException.

Results – Data Figure 4 shows that the NonBlockingQueueTakeOfferAction clearly outperform the BlockingQueueTakeOfferAction.

Analysis – In this test, we totally defer synchronization implementation to the underlying Java objects. We see that in the scenario where there are an equal number of puts and takes from the queue, the ConcurrentLinkedQueue is the preferred choice over LinkedBlockingQueue. Of course, these classes do not implement the same interfaces and, as such, do not have identical APIs. In the case where some functionality that only LinkedBlockingQueue provides, such at put or remove, is required the developer may not have a choice.

High-Powered Machine Results

All tests listed above were run in a second environment with considerably stronger hardware. This machine had 16 x Quad-Core Opteron 8378 processors, 125GB RAM, and a Gnu Linux OS. Identical test showed negligible differences between synchronization models in each case. The power of the machine marginalized advantages that any of the models may have had. This testing also highlighted one of the drawbacks of the measurement model. When taking average times, a test can run indefinitely. In the current implementation, where a counter is constantly incremented, there is an upper bound on the number of actions taken that is constrained by the maximum value of a Long object in Java. If testing requires, higher values can certainly be accommodated by adding smarter counting logic that represents numbers higher than the max Long value.

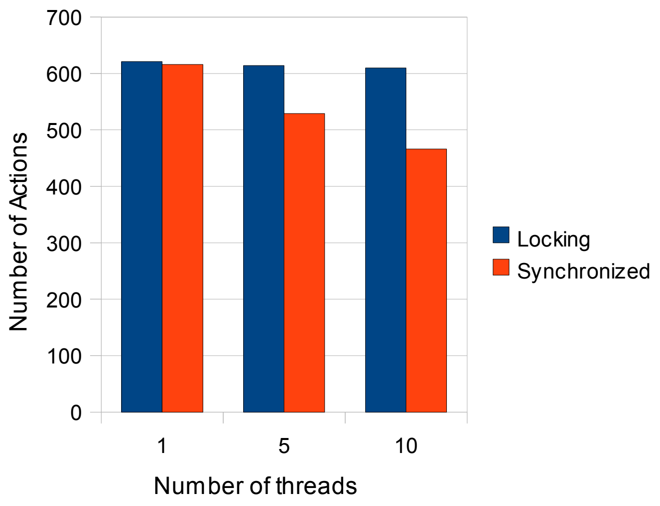

Figure 1: Contains Algorithms

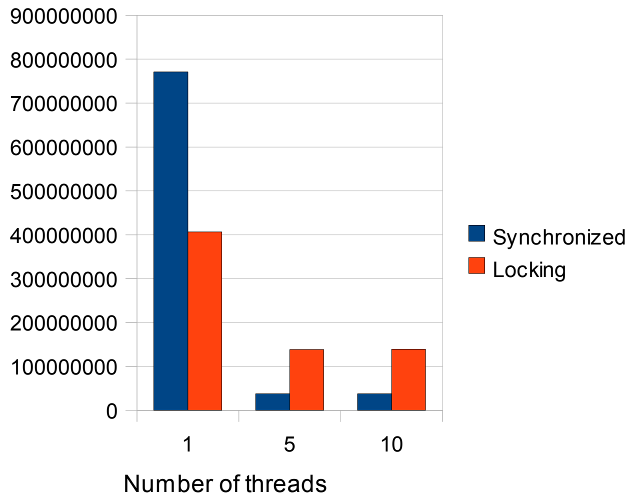

Figure 2: Simple Loop Algorithms

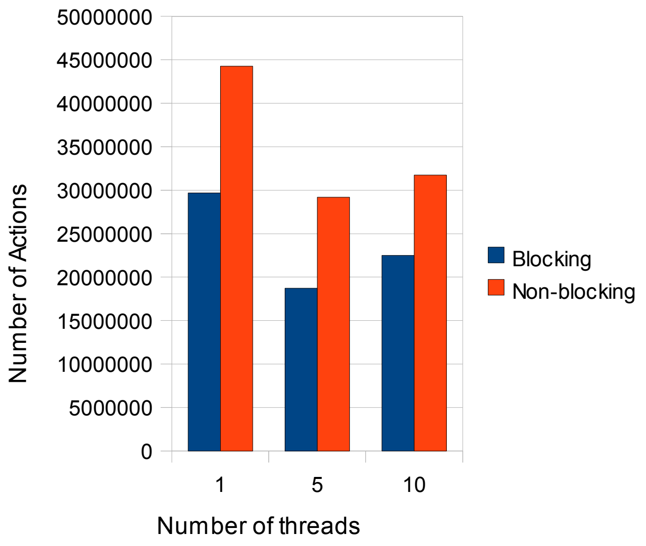

Figure 4: Queue Algorithms

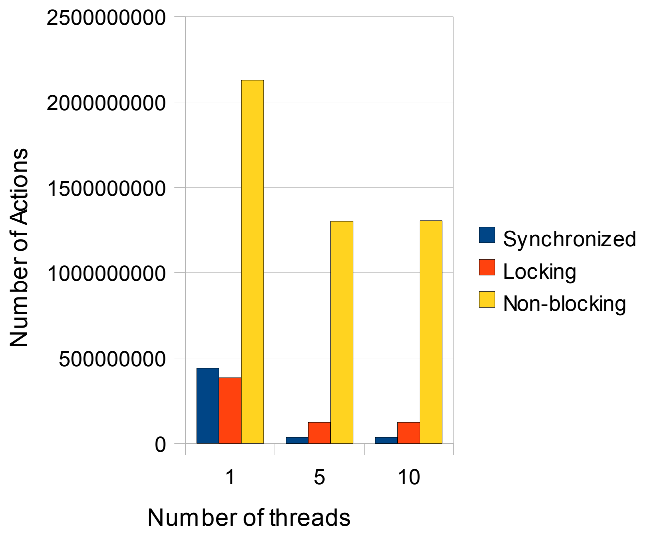

Figure 3: Counter Algorithms

Chapter 5: Discussion of the Framework

This section provides an assessment of the framework against the goals stated above. Flexibility proved to be the most valuable area when executing tests and, fortunately, an area in which the framework showed very well. Meeting this goal tied in tightly to the ability to provide configurability, which came at a predictably low cost of effort. Many run-time properties were extracted as properties, including the number of test runs, length of test execution, number of threads, and set of tests to run. When comparing environments, interesting results occurred that prompted more investigation. The flexibility of the test engine was valuable in drilling down on the different variables by rerunning tests with varying parameter such as the number of threads or execution time.

The framework also performed admirably in the area of usability. The setup overhead to create new test using combinations of existing and new Action and Actor could be measured in minutes. The CSV format for the output allowed for easy translation in to visualization tools.

It is challenging to measure the actual impact of the measurement constructs, by nature, since this only adds to the risk of Observer Effect. Still, it is safe to assume the impact of the measurement in this case is negligible. The choice to measure in terms of actions taken allowed for an aggregate view that didn't depend on the accuracy of measuring the local action. Accordingly, the overhead for a given test came only in the form of incrementing an action counter.

Regarding the goal of being concurrency oriented, this was accomplished through the separation of Action and Actor modules. This lead to a natural division between what was being synchronized and how it was being synchronized when defining the tests. In the design phase, there was thought given to tailoring the framework closer to low level acquire and release operations commonly used in synchronization research. This was abandoned, however, when it became clear that this did not map well to non-blocking algorithms.

Future enhancements

Several ideas for future features arose out of this exercise. Enhanced reporting is a natural progression. Currently, the output data and format are fixed. It may be useful to allow for reporting more granular results, the output to additional formats such as html, or the output of more complex measures such as standard deviation.

Another potential direction would be to adopt the framework so that it can attach to existing code. Ideally, the framework would be able to dynamically adjust implementations based on actual usage patterns.

Challenges

The largest roadblock came in the area of comparing JDKs. The framework must be written in the lowest version of the JDK under test or, in this case, rewritten. This was not an critical block since the framework was so lightweight. In the case of real world deployment, however, there would rarely be the need to test against retired versions of the JVM. The other major challenge was to determine the proper tradeoff between precision of the results against the impact of the measurement logic. In the end, the balance shifted more towards minimal impact since the goal was not the precise measurement of individual synchronization blocks, but a better understanding of aggregate performance.

Chapter 6: Summary

This paper outlined a framework for testing the performance of concurrent algorithms implemented in Java. It went on to vet the framework through the execution of a series of tests aimed at better understanding synchronization options in Java and the commonly held beliefs about them. The disparate results from executing these tests across environments highlighted the need for such a portable, repeatable, and flexible test framework. Preliminary efforts suggest that further exploration of the framework should provide expanded understanding of the run time performance of multi-threaded Java applications.

Appendix

Abstract Action:

```
public abstract class AbstractAction implements Action {
	protected int delay;
	public AbstractAction() {
		super();
	}
	public int getDelay() {
		// TODO Auto-generated method stub
		return 0;
	}
	public void setDelay(int delay) {
		this.delay = delay;
	}
	public boolean executeAction(ActionArgument[] argument) {
		if (delay > 0) {
			try {
				Thread.sleep(delay);
				return executeActualAction();
			} catch (InterruptedException e) {
				e.printStackTrace();
				return executeActualAction();
			}
		} else {
			return executeActualAction();
		}
	}
	protected abstract boolean executeActualAction();
}
```

Simple Loop Action:

```
public class SimpleLoopAction extends AbstractAction {
	private static final int LOOP_SIZE = 1000000;
	public SimpleLoopAction() {
		super();
	}
	protected boolean executeActualAction() {
		int count = 0;
		for (int i=0; i < LOOP_SIZE; i++) {
			count++;
		}
		return true;
	}
}
```

List Contains Action:

```
public class ListContainsAction extends AbstractAction {
	private static final int LIST_SIZE = 1000000;
	private static List<String> dummyList = new ArrayList<String>(LIST_SIZE);
	private final String value;

	public ListContainsAction() {
		super();
		value = "dummyValue";
		for (int i=0; i < LIST_SIZE; i++) {
			dummyList.add(Integer.toString(i));
		}
	}
	protected boolean executeActualAction() {
		dummyList.contains(value);
		return true;
	}
}
```

Atomic Counter Action:

```
public class AtomicCounterAction extends AbstractAction {
	private int value;
	public AtomicCounterAction() {
		super();
	}
	protected boolean executeActualAction() {
		value++;
		 return true;
	}
}
```

Non Atomic counter Action:

```
public class NonAtomicCounterAction extends AbstractAction {
	private AtomicInteger value;
	public NonAtomicCounterAction() {
		super();
		value = new AtomicInteger(0);
	}
	protected boolean executeActualAction() {
		int v;
		v = value.get();
		return value.compareAndSet(v, v + 1);
	}
}
```

Blocking Queue Take/Offer Action:

```
public class BlockingQueueTakeOfferAction extends AbstractAction {

	private static final int LIST_SIZE = 1000000;
	private static Queue<String> queue = new LinkedBlockingQueue<String>(
			LIST_SIZE);
	private final String value;
	public BlockingQueueTakeOfferAction() {
		super();
		value = "dummyValue";
		for (int i = 0; i < LIST_SIZE; i++) {
			queue.add(Integer.toString(i));
		}
	}
	protected boolean executeActualAction() {
		String temp = queue.remove();
		queue.offer(temp);
		return true;
	}
}
```

Blocking Queue Take/Offer Action:

```
public class NonBlockingQueueTakeOfferAction extends AbstractAction {
	private static final int LIST_SIZE = 1000000;
	private static Queue<String> queue = new ConcurrentLinkedQueue<String>();
	private final String value;
	public NonBlockingQueueTakeOfferAction() {
		super();
		value = "dummyValue";
		for (int i = 0; i < LIST_SIZE; i++) {
			queue.add(Integer.toString(i));
		}
	}
	protected boolean executeActualAction() {
		String temp = queue.remove();
		queue.offer(temp);
		return true;
	}
}
```

Actor:

```
public abstract class Actor extends Thread {
	protected final int id;
	private static AtomicInteger idCounter = new AtomicInteger(0);
	protected long actionCount;
	protected final Action action;
	public Actor(Action action) {
		this.action = action;
		id = idCounter.incrementAndGet();
		init();
	}
	protected abstract void act();
	protected abstract void init();
	public abstract void finish();
	public String getActorId() {
		return Integer.toString(id);
	}
	public long getActionCount() {
		return actionCount;
	}
	public void reset() {
		actionCount = 0;
	}
	public void run() {
		act();
	}
	public Actor clone() {
		Class<Actor> c = (Class<Actor>)this.getClass();
		try {
			Actor saNew = null;
			Constructor<Actor>[] constructors =
				(Constructor<Actor>[])c.getConstructors();
			if (constructors.length > 0) {
				for (int index=0; index < constructors.length; index++) {
					Constructor<Actor> constructor =
						constructors[index];
					try {
						saNew =
						(Actor)constructor.newInstance(this.action);
					} catch (IllegalArgumentException e) {
						e.printStackTrace();
						continue;
					} catch (InvocationTargetException e) {
						e.printStackTrace();
```

```
                        continue;
                    }
                    break;
                }
            }
            return saNew;
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        return null;
    }
    protected void log(String msg) {
        System.err.println("Actor<" + id + "> : " + msg);
    }
}
```

Locking Actor:

```
public class LockingActor extends Actor {
	private boolean shouldRun = true;
	private static final Lock lock = new ReentrantLock();
	public LockingActor(Action action) {
		super(action);
	}
	protected void act() {
		while (shouldRun) {
			try {
				lock.lock();
				action.executeAction(null);
				actionCount++;
			} finally {
				lock.unlock();
			}
		}
	}
	protected void init() { }
	public void finish() {	shouldRun = false; }
}
```

Synchronized Actor:

```
public class SynchronizedActor extends Actor {
	private static Object lock = new Object();
	private boolean shouldRun = true;
	public SynchronizedActor(Action action) {
		super(action);
	}
	protected void act() {
		while (shouldRun) {
			synchronized(lock) {
				action.executeAction(null);
			}
			actionCount++;
		}
	}

	@Override
	public void finish() {	shouldRun = false; }
	protected void init() { }
}
```

Non-blocking Actor:

```
public class NonBlockingActor extends Actor {
        private boolean shouldRun = true;
        public NonBlockingActor(Action action) {
                super(action);
        }
        protected void act() {
                while (shouldRun) {
                        while (!action.executeAction(null)) {
                                action.executeAction(null);
                        }
                        actionCount++;
                }
        }
        protected void init() { }
        public void finish() {
                shouldRun = false;
        }
}
```

Unsynchronized Actor:

```
public class UnSynchronizedActor extends Actor {
        private boolean shouldRun = true;
        public UnSynchronizedActor(Action action) {
                super(action);
        }
        protected void act() {
                while (shouldRun) {
                        action.executeAction(null);
                        actionCount++;
                }
        }
        protected void init() { }
        public void finish() {
                shouldRun = false;
        }
}
```

Bibliography

1. Doug Lea, “The java.util.concurrent synchronizer framework”, *Science of Computer Programming* 58 (2005) : 293-309.
2. JSR 166 Expert Group. Jsr-166: Concurrency Utilities. http://jcp.org/en/jsr/detail?id=166, September 2004.
3. Brian Goetz, *Java Concurrency in Practice* (Upper Saddle River: Addison-Wesley, 2006).
4. Rob Pooley, “Software Engineering and Performance: A Road Map” *In Proceedings of the Conference on the Future of Software Engineering* (2000) : 189-199.
5. Filippos I. Vokolos, “Performance Testing of Software Systems”, *Workshop on Software and Performance* (1998): 80-87
6. Yaniv Eytani, “Towards a framework and a benchmark for testing tools for multi-threaded programs”, *Concurrency and Computation: Practice and Experience* 19 (2007) : 267–279
7. Arkady Bron, “Applications of synchronization coverage”, *Principles and Practice of Parallel Programming* (2005): 206-212.
8. Orit Edelstein, “Framework for testing multi-threaded Java programs”, *Concurrency and Computation: Practice and Experience* 15 (2003) : 485-499.
9. Maged Michael, “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms”, *Annual ACM Symposium on Principles of Distributed Computing* (2006) : 267-275.
10. Joel Emer, “Asim: A Performance Model Framework”, *Computer* 35 2 (2002) : 68-76.