Multi-Core in JAVA/JVM

Tommi Zetterman

Concurrency Prior to Java 5: Synchronization and Threads

Java has been supporting concurrency from the beginning. Typical Java execution environment consists of Java Virtual Machine (JVM) which executes platform-independent Java bytecode. JVM is run as a single process, and Java application can launch multiple threads to implement concurrency.

Java memory model describes how concurrent threads interact through memory. The processor and the memory subsystem is free to introduce any useful execution optimization as long as the result of executing the thread is guaranteed to be exactly the same it would be if thread statements have been executed in the program order. When considering multiple threads executed concurrently, Java memory model defines that the actions which happen prior to communication between threads (locking, releasing lock) are seen by other threads. If Java VM is optimized so that it works with a local cached copy of data, it must make its changes to the shared data visible to other threads after releasing a lock.

As said, concurrency in Java is based on threads and synchronization between them. Java has a *Thread* class, which has methods to run thread code run and synchronize between other threads. Some commonly used Thread methods are:

public void run()

- The body of thread

public synchronized void start()

- Starts the thread and invokes run() method

public final synchronized void join(long milliseconds)

- Wait for this thread to die (optionally a time-out can be specified)

public static void yield()

- Thread yields processor to some other runnable Thread

public final int getPriority() / public final void setPriority(int newPriority)

- return / set threads priority

The most convenient way to implement own threads is to extend the existing Thread class. As a drawback, this approach cannot be used if the own thread should extend some other class because Java does not support multiple inheritance. In that cases, programmer can define a class implementing *Runnable* interface. The listning below illustrates both ways; MyThread1 is done by implementing Runnable interface, MyThread2 is done by subclassing Thread.

```
public class MyThread1 implements Runnable {
    public void run() {
     // thread code
    }
    public static void main(String args[]) {
        (new Thread(new MyThread1())).start();
```

```
          }
      }
      public class MyThread2 extends Thread {
      public void run() {
              //thread code
          }
          public static void main(String args[]) {
              (new MyThread2()).start();
          }
      }
```

Before version 5, Java provides two ways for synchronization between threads, *synchronized methods* and *synchronized statements*. If class has synchronized methods, it means that only a one thread at a time can run a synchronous method for object instantiated from the class, other threads must wait until the first thread is done. When a synchronous method is exited, the changes made by the first thread to the object state become visible to other threads (happens before -relationship). When a thread enters to the synchronous methods, it acquires an implicit mutex, which is released when the thread exists the synchronous method. The implicit mutex is owned by the thread, which allows synchronous method to be called recursively. For example, synchronous methods can be used to implement concurrent counter:

```
public class SynchronizedCounter {
  public synchronized void update(int x) {
    count += x;
  }
  public synchronized void reset {
    count = 0;
  }
}
```

Synchronized statements can be used when a finer grained synchronization. As critical section in synchronous statements can be any code block in method, they allow critical sections to be smaller. Explicit definition of used lock object allows two noninteracting synchronous statements of the same object to be executed concurrently. The use of synchronous statements and methods may hurt performance even when there's no collisions. This happens when exiting a synchronous statement causes possible cached copies where threads make their data to be flushed.

The example how to use synchronous statements is shown in the following code block.

```
public class MsLunch {
  private long c1 = 0;
  private long c2 = 0;
  private Object lock1 = new Object();
  private Object lock2 = new Object();

  public void inc1() {
     synchronized(lock1) { c1++; }
  }
```

```
    public void inc2() {
     synchronized(lock2) { c2++;  }
    }
  }
```

Java 5

Java 5 was released in 2006 . It improves the concurrent programming by providing constructs for finer-grained synchronization, more precise and low level synchronization control and performance increase. In addition, it introduced some concurrent containers for easy use. Concurrency support introduced in version 5 was clearly targeted to parallel execution enabled by multi-core CPUs by allowing better performance and more detailed synchronization control.

This chapter introduces the contents of the tree added concurrency support packages

- *java.util.concurrent* defines utility classes commonly use in concurrent programming
- *java.util.concurrent.atomic* defines a small toolkit of classes which support a lock-free thread-safe programming on single variables.
- *java.util.concurrent.locks* defines interfaces and classes for locking and waiting for certain condition. This package allows a creation of own synchronization frameworks different than built-in locking and monitors.

Atomic Objects

Package java.util.concurrent.atomic introduces atomic variables which can be mutated by exploiting CAS (compare-and-swap) -like instruction available in most modern CPU architectures. When an atomic object is accessed, the operation either happens completely, or not at all. No side effects are visible until the operation is completed. These constructs can be used for example to build lock-free data structures. Package contains single-variable classes, like *AtomicBoolean*, *AtomicInteger*, *AtomicReference* and *AtomicLong*. Example some methods *AtomicInteger* provides are:

int addAndGet(int delta)

- Atomically add given value to current value

boolean compareAndSet(int expect, int update)

- Atomically set the value to update if old value equals to expect

int incrementAndGet()

- Atomically ingrement current value by one

int decrementAndGet()

- Atomically decrement current value by one

For example the following *Sequencer* provides a sequence of number to the concurrent threads:

```
class Sequencer {
     private AtomicLong sequenceNumber = new AtomicLong(0);
     public long next() {
          return sequenceNumber.getAndIncrement();
     }
}
```

Lock Objects

Package java.util.concurrent.locks can be used to build own synchronization mechanism instead of using build-in synchronous methods or synchronous statements. It allows programmer a greater flexibility in the use of locks. It defines three interfaces:

- *Condition* defines methods allowing thread to wait until it is signaled. When a tread starts waiting, it frees locks atomically and signals other thread(s) to wake up. This allows to create a wait/notify mechanism which may be useful for example in concurrent FIFO. And an example, implementation of put():

```
final Lock lock = new ReentrantLock();
final Condition notFull  = lock.newCondition();
final Condition notEmpty = lock.newCondition();

 public void put(Object x) throws InterruptedException {
   lock.lock();
   try {
     while (count == items.length)
     notFull.await();
     items[putptr] = x;
     if (++putptr == items.length) putptr = 0;
     ++count;
     notEmpty.signal();
   } finally {
     lock.unlock();
   }
 }
```

- *ReadWriteLock* interface defines locks that may be shared among multiple readers, but are exclusive to writers. ReadWriteLock maintains a pair of locks, one for read-only operations and one for writing. The read-only lock can be acquired simultaneously by multiple threads, while the write lock is exclusive. It allows a higher level of concurrency by allowing multiple concurrent reader threads to simultaneously gain a read-only access to the shared data.

- *Lock* provides more extensive locking than synchronized methods or synchronized statements. It allows the locking and unlocking to be made in different scopes, locking and releasing of multiple locks in the desired order, attempting if a lock can be acquired (trylock) and interruptable and timed locks. While providing greater flexibility to the programmer, using *Lock* objects also requires greater responsibility. Locks are not released automatically like in block structure -based automatic locking used with synchronous methods and synchronous statements, but the programmer has to take care of it. The following idiom is suggested to be used with *Lock* objects to ensure that lock is released:

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

Executor Framework

Package java.util.concurrent defines the executor framework which allows programmer to create his own thread management and scheduling scheme. Executor separates task execution and submission, and provides hooks for thread life-cycle monitoring (e.g. statistics gathering) and control. The main interfaces are:

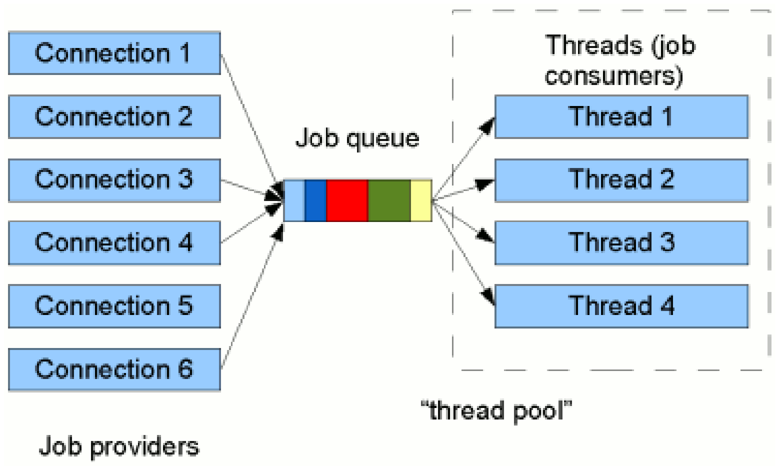
- *Callable<V>* interface is similar to Runnable except it has a call() method to be able to return a value
- *Future<V>* represents the result of asynchronous computation. There are methods to check if computation is completed, wait for it to be finished, get the results of computation, and cancel the computation.
- *Executor* is an interface for object to execute submitted tasks. This interface decouples the task submission from the task execution, and it is used instead of creating threads explicitly. For example, in the following code the DirectExecutor executes tasks in serial, and ThreadPerTaskExecutor executes every task in separate thread. Task submission is done similarly in both cases.

  ```
  class DirectExecutor implements Executor
  {
       public void execute(Runnable r) {
           r.run();
       }
  }
  class ThreadPerTaskExecutor implements Executor {
       public void execute(Runnable r) {
                 new Thread(r).start();
        }
  }
  ```

- *ExecutorService* is an Executor that provides methods to manage the termination (shut down), task tracking and returning the *Future* of task which can be used to cancel execution, wait for completion, track the progress of the task.
- *ScheduledExecutorService* is an ExecutorService which allows to schedule commands to be run after a delay, or to be executed periodically. For example, BeeperControl illustrates the use of ScheduledExecutorService. It creates two Runnable tasks, the first one which writes "beep" periodically with a 10 seconds period, and the second one which cancels the first one after an hour.

```
class BeeperControl {

  private final ScheduledExecutorService scheduler =
    Executors.newScheduledThreadPool(1);

  public void beepForAnHour() {

    final Runnable beeper = new Runnable() {
     public void run() { System.out.println("beep"); }
    };

    final ScheduledFuture<?> beeperHandle =
      scheduler.scheduleAtFixedRate(beeper, 10, 10,  SECONDS);
      scheduler.schedule(new Runnable() {
        public void run() { beeperHandle.cancel(true); }
            }, 60 * 60, SECONDS);
  }
}
```

Another example of ExecutorService is the *ThreadPoolExecutor*. It is an ExecutorService which executes submitted task by using one of the pooled threads, as depicted in the following figure. It allows the re-use of threads to execute several tasks, which increases the performance especially when there is a large number of small asynchronous tasks to be performed. When the thread already exists and is started, the cost related to the invocation of a new thread is saved. Job queue is used transfer and hold submitted tasks. Any BlockingQueue can be used as a job queue.



Queues

Package java.util.concurrent defines several queues meant for thread-safe use.

- *BlockingQueue<E>* interface defines a thread-safe queue. It is mainly meant for producer-consumer queues, but supports also the Collection interface (although adding and removing arbitrary elements is usually not efficient). BlockingQueue also supports operations to wait queue to become nonempty when retrieving an element, and wait for space to become available in queue when storing an element. Classes implementing BlockingQueue are *LinkedBlockingQueue* which is unbounded and based on linked nodes, *ArrayBlockingQueue* which is based on array to store elements and which capacity cannot be increased after created, *SynchronousBlockedQueue* which capacity is 1 and every put must wait for a take (and vice versa), *PriorityBlockingQueue* which stores the elements in some order, and *DelayedQueue* which allows elements to be taken out only after its specific delay has expired.
- *ConcurrentLinkedQueue<E>* interface defines an unbound, non-blocking thread-safe queue based on linked nodes.

Synchronizers

Package java.util.concurrent defines four synchronization constructs

- *Semaphore* is a counting semaphore, allowing a restricted number of threads to access a shared resource. Each *acquire* decrements permit and blocks if necessary until the resource is available. Each *release* increments permit.
- *CountDownLatch* allows to wait until a set of operations is completed. It is initialized to a given value, and the *countDown()* method decrements it atomically. The a*wait()* method blocks until CountDownLatch reaches zero.
- *CyclicBarrier* allows to create a barrier for a set of treads. Threads have to wait each others to reach a common barrier point before continuing. “Cyclic” means that barrier can be reused after waiting threads are released.
- *Exchanger* allows to create a synchronization point for two threads to exchange objects.

Concurrent Collections

Package java.util.concurrent defines thread-safe collections to be used by multiple threads.

- *ConcurrentyHashMap* is a hash table which supports full concurrency for retrievals (get) and adjustable concurrency for updates (put, remove). Parameter *concurrency level* can be used to adjust the performance of concurrency updates versus the size of hash table. A too big concurrency level increases the size and decreases the performance, while a too small concurrency level causes contention for updates. A good first guess for a concurrency level is the number of simultaneously writing threads.
- *CopyOnWriteArrayList* is a thread-safe ArrayList where all mutable operations are implemented by making a fresh copy of the array. While this can make the mutation very costly, the benefit is that there is no need to synchronize readers. This collection may be a good choice when read operations (e.g. traversal) vastly outnumber mutations.

- *CopyOnWriteArraySet* is a Set implemented similarly to CopyOnWriteArrayList (all mutable operations make a fresh copy).

Concurrency in Java 7

Since the introduction of Java 5, the number of cores in multicore processors has increased. The hardware trend is that in future the number of cores will continue to increase, so finer grained and more scalable parallelism is needed to keep all processor cores busy. To supports this trend, Java 7 introduces to things: *fork-join framework* to exploit divide-and-conquer parallelism, and *ParallelArray* containing high-level parallel collections.

Fork-Join Framework

Fork-joint framework is targeted for divide-and-conquer style algorithms that break up the work recursively until it is small enough to be computed sequentially. The following pseudocode illustrates the approach. The INVOKE-IN-PARALLE -step waits for the both halves to be complete. For example, merge sort and binary search are algorithms easily exploiting fork-join parallelism.

```
// PSEUDOCODE
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```

Fork-join parallelism support in Java 7 is implemented in package java-util.concurrent.forkjoin. It defines a *ForkJoinTask* which is a light-weight thread. For example, I/O and blocking synchronization for ForkJoinTasks are not allowed, and the task should be independent of other ForkJoinTasks. *ForkJoinPool* hosts a *ForkJointExecutor* which is optimized to execute ForkJointasks.

To balance the computation load between threads, the worker threads use *work-stealing*. Each worker thread has it owns working queue which is implemented by using a double-ended queue (dequeue). When a thread forks a new ForkJoinTask, it is pushed into the head of threads dequeue. When the thread executes a join operation with another task that has not yet completed, rather than sleeping until the target task is complete, it pops another task off the head of its dequeue and executes it (LIFO). In the case when the threads task queue is empty, it tries to steal a task from the tail of another thread's dequeue (FIFO).

Work stealing automatically corrects unequal distribution of work without central coordination. Because the work is stolen from the tail of the queue, the stolen task is usually “big”, which tries to minimize the total number of steal events.

As an example, the following MaxSolver illustrates how to use fork-join framework. Because different subproblems of MaxSolver work on disjoint portions of the data, data copying is not needed, and the implementation is efficient.

```
class MaxSolver extends RecursiveAction {
  private final MaxProblem problem;
    int result;

  protected void compute() {
    if (problem.size < THRESHOLD)
      result = problem.solveSequentially();
    else {
      int m = problem.size / 2;
      MaxSolver left, right;
      left = new MaxSolver(problem.subproblem(0, m));
      right = new MaxSolver(problem.subproblem(m,problem.size));
      forkJoin(left, right);
      result = Math.max(left.result, right.result);
     }
  }
}

ForkJoinExecutor pool = new ForkJoinPool(nThreads);
MaxSolver solver = new MaxSolver(problem);
pool.invoke(solver);
```

Performance

The following table shows the measured performance of select-max for 500k-element array on various systems with various sequential threshold parameters. The performance increase is significant as long as the sequential threshold is somehow reasonable. Code is independent of the execution topology, so the performance gain is portable for different CPUs

Threshold=	**500k**	**50k**	**5k**	**500**	**50**
Pentium-4 HT (2 threads)	1.0	1.07	1.02	0.82	0.2
Dual-Xeon HT (4 threads)	0.88	3.02	3.2	2.22	0.43
8-way Opteron (8 threads)	1.0	5.29	5.73	4.53	2.03
8-core Niagara (32 threads)	0.98	10.46	17.21	15.34	6.49

Parallel Array

Java 7 specifies aggregate operations on arrays to be automatically executed by using the fork-join decomposition. Parallel arrays increase the abstraction level, because programmer does not need to explicitly write fork-join code. Parallel operations support

- *Filtering* which means selecting the subset of the elements in array. It is possible to use multiple filters.Binary search is supported on sorted parallel arrays.

- *Mapping* converts selected elements to another form
- *Replacement* is used to create a new parallel array derived from the original one
- *Aggregation* is used to combine all values into a single value like max, min, sum, average (general-purpose reduce method)
- *Application* is used to perform an action for each selected element.

For example, the following example shows how ParallelArray filtering, mapping and aggregation operations are used to query a senior student with the best average grade points.

```
ParallelArray<Student> students = new
              ParallelArray<Student>(fjPool, data);
double bestGpa = students.withFilter(isSenior)
                         .withMapping(selectGpa)
                         .max();

public class Student {
    String name;
    int graduationYear;
    double gpa;
}

static final Ops.Predicate<Student> isSenior = new
Ops.Predicate<Student>() {
    public boolean op(Student s) {
        return s.graduationYear == Student.THIS_YEAR;
    }
};

static final Ops.ObjectToDouble<Student> selectGpa = new
Ops.ObjectToDouble<Student>() {
    public double op(Student student) {
        return student.gpa;
    }
};
```

Performance

The following table represents the results of measuring performance for the max_GPA query in Core 2 Quad system running windows. As expected, the best speedup is achieved when the number of cores equals to the number of threads.

Table 1. Performance measurement for the max-GPA query
(Core 2 Quad system running Windows)

Threads / **Students**	**1**	**2**	**4**	**8**
1000	1.00	0.30	0.35	1.20
10000	2.11	2.31	1.02	1.62
100000	9.99	5.28	3.63	5.53
1000000	39.34	24.67	20.94	35.11
10000000	340.25	180.28	160.21	190.41

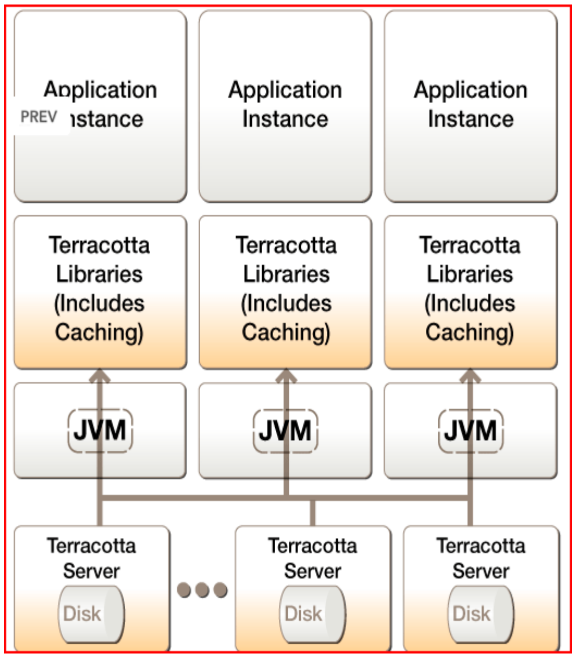
Other topics

Terracotta

Terracotta is an open source infrastructure software to scale Java application to many computers. It makes a multi-threaded program clustered without additional code. Threads to be executed in cluster machines are specified in .xml file.

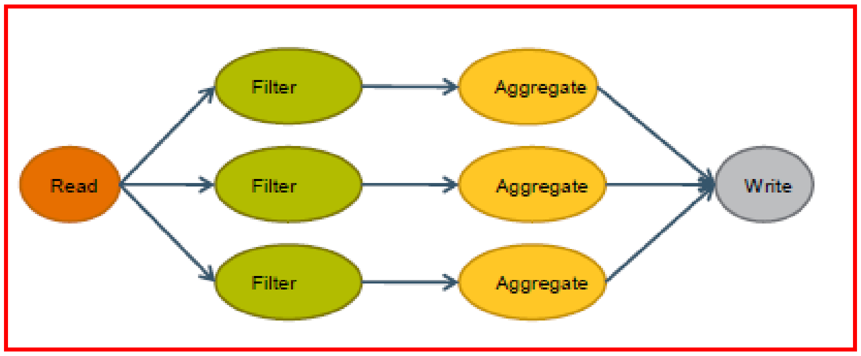
To run terracotta application on multiple machines, a terracorra server is installed to hosts and started. Then, on client side, terracotta servers addresses are specified. Terracotta provides a persistent clustered heap between multiple machines.

Terracotta does not include load balancer. To minimize unnecessary data traffic between hosts, it is adviced to use a load balancing scheme that supports sticky sessions.



Stream Programming

Pervasive DataRush is a stream programming solution for Java, targeted for data-intensive applications Computing model is based on dataflow graphs consisting of nodes representing computation tasks, and directed arcs representing interconnect queues. Programmer defines the computation made in nodes, and the flow of data between nodes by wiring node outputs to inputs. Ones composed, the resulting dataflow model can be executed by invoking a run() method for graph. Pervasive DataRush framework takes care of synchronization and thread management.



Highly Scalable Lib

Highly scalable lib is a replacement for java.util.* and java.util.concurrent.* collectionsproviding better performance and scalability for a huge number of cores. Scalability was tested to be linear up to 768 cores. As an example, it contains:

- *ConcurrentAutotable* which is a auto-resizing table of longs supporting low-contention CAS operation. The intention is to support highly scalable structures (e.g. counter) where uphappen at so high volume that the cache contention for CAS'ing a single word is unacceptable. Updates are done with CAS to no particular table element. Hash function is used to select updated element to spread updates evenly.
- *NonBlockingHashMap* is a lock-free implementation of ConcurrenHashMap
- *NonBlockingSetInt* is a lock free bit vector set.

Transactional memory

Transactional memory is a programming paradigm where the sequences of memory operations are grouped as *transactions*, which are either executed completely or have no effect at all. To define transactions, programming language can be added an *atomic* statement:

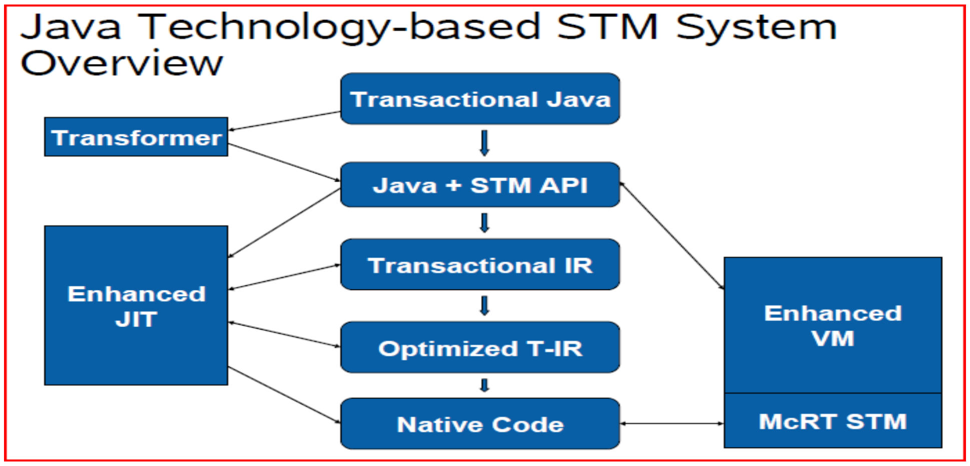
```
atomic {
     if (inactive.remove(p))
     active.add(p);
}
```

When accessing a shared memory, no locking is needed. If no conflicts occur during the access, the changes transaction made are made visible outside (commit). If there are conflicts (either two or more transactions updates the shared data, or one transaction has updated the data the other transaction is reading before the reading transaction has completed), one or more transactions are aborted (perhaps to be retry later).

Transactional memory may be implemented either in software (STM) or in hardware (HTM). HTM typically uses cache coherence protocol for conflict detection, so the conflict detection granularity is one cache line. In case of STM, there's more freedom, granularity may be for example one word or one object.

Transactional memory way provide weak or strong isolation, meaning that transaction are executed in isolation relative to other transactions only, or relative to both transactional and non-transactional accesses, respectively. Data management may be based keeping the log about changes to be able to roll-back in case of abort (direct updated) or making the changes to private copy which is made visible outside of the transaction during commit (deferred update).

McRT STM is Java-based STM system which enhances Java VM to contain run-time support for transactions and internal data structures for transactional records. Transactional Java is compiled to Java+STM API which inserts barriers to memory access, access tracking and conflict detection. Control is modeled by using Java exception handling. Java+STM API code is compiled further to internal presentation (IR) and then to bytecode, which maps STM SPI to JIT IR opcodes.



References

- `http://jcp.org/en/jsr/detail?id=166`
- `http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html#package_description`
- `http://www.multicoreinfo.com/2008/10/jvm-challenges-and-directions-in-the-multicore-era/`
- `http://developers.sun.com/learning/javaoneonline/j1sessn.jsp?sessn=TS-5515&yr=2008&track=javase`
- `http://www.ibm.com/developerworks/java/library/j-jtp03048.html?ca`
- `http://en.wikipedia.org/wiki/Java_version_history#Java_SE_7`
- `http://sourceforge.net/projects/high-scale-lib`
- `http://www.terracottatech.com`
- `http://www.eecg.toronto.edu/~amza/ece1747h/papers/mcrt.pdf`
- `http://developers.sun.com/learning/javaoneonline/2008/pdf/TS-6316.pdf`