A Report on:

Energy efficient algorithm design in Hadoop clusters

By

Kartik Sathyanarayanan
(2013A7PS037G)

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI,
K.K BIRLA GOA CAMPUS

Introduction

Hadoop is an open-source implementation of MapReduce enjoying wide adoption and is often used for short jobs where low response time is critical. Hadoop's performance is closely tied to its task scheduler, which implicitly assumes that cluster nodes are homogeneous and tasks make progress linearly, and uses these assumptions to decide when to speculatively re-execute tasks that appear to be stragglers. In practice, the homogeneity assumptions do not always hold. MapReduce uses speculative execution to improve fault tolerance. Current Hadoop implementation decides whether to run speculative tasks based on the progress rates of running tasks, which does not take into consideration the absolute progress of each task.

The modified Hadoop framework was deployed in 6 t2.medium EC2 instances in a master-slave configuration.

Modified Task Scheduler

The BASE scheduler (which itself was a little improvement from LATE scheduler) was slightly modified to limit unnecessary speculative executions.

The need for speculating a task is determined by the following factors:

- The task should not be done or almost done.
- The task should not have been speculatively executed already.
- Launching a speculative task should not exceed the upper limit on maximum number of currently active tasks.
- The task should have a burn in period – this period is used to obtain usable task-specific CPU metrics for estimating speculation cost.
- The version of HDFS used cannot support multiple simultaneous writes. As a restrictive safety measure, map tasks are not speculated.
- A weighted factor in estimating the speculation benefit is (estimated completion time – speculated completion time) [Nidhi's proposal]. This difference is estimated as the difference mean completion time of the other tasks and the current progress rate (based on the LATE/BASE scheduler ideas) [2].

Ultimately, the decision of speculating a task is taken by checking if the calculated speculation benefit is higher than a threshold. This threshold is decided by a multiple of the standard deviation of the completion time of tasks. Empirically, this is between 2 * standard deviation and 3 * standard deviation. This multiple is parameterized as the slow task threshold[1].

Tasks which pass this filtering process are added to a list. This list is then sorted based on the speculative gain (ratio). The estimated time to completion is measured [1] as the percentage of task left to complete – that is, (1 – progress) / (progress rate of the task). The one with the highest speculative gain is then sent for speculation.

Algorithm

- SpeculativeCap – The max fraction (0-1) of running tasks that can be speculatively re-executed at any time. We have used a value of 1.
- SlowTaskThreshold – The number of standard deviations by which a task's average progress-rates must be lower than the average of all running tasks' for the task to be considered too slow. We have used a value of 0.01.
- SlowNodeThreshold – The number of standard deviations by which a Task Tracker's average map and reduce progress-rates must be lower than the average of all successful map/reduce task's for the TT to be considered too slow to give a speculative task to. We have used a value of 0.01.
- If a task slot becomes available in a node and there are less than SpeculativeCap(fraction) speculative tasks running :
    - Ignore the request if the TaskTracker's progressRate < SlowNodeThreshold
    - Choose candidate tasks : those tasks whose progress rates are below [slowTaskThreshold * mean(progress rates)] - *Nidhi's proposal*
    - Rank the candidate tasks that are not currently being speculated by estimate time left
    - Speculate the task that's expected to complete last

Benchmarks

1. Unmodified Hadoop 0.20

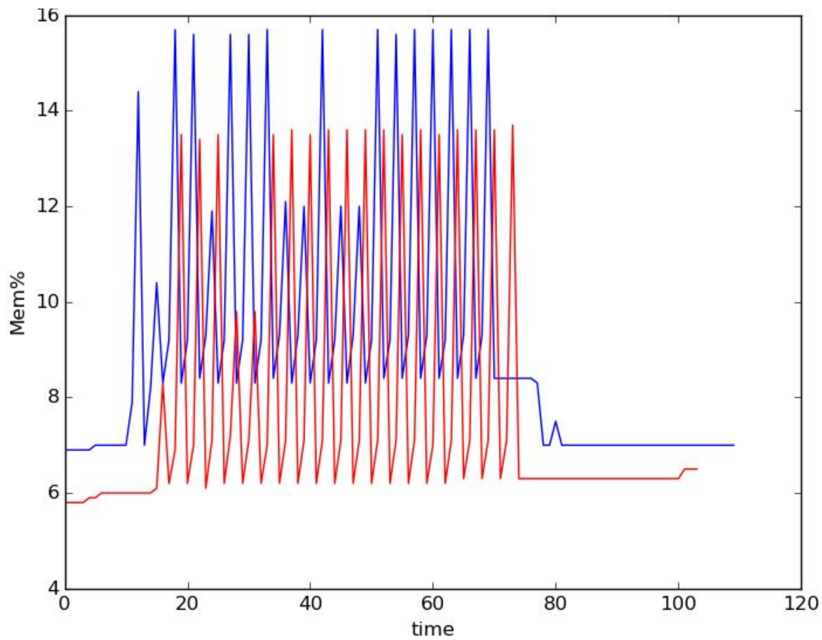
Job Name	Command Line arguments	Job Response time	No. of maps	No. of reduces	Speculative maps launched	Speculative reduces launched
PiEstimator	200, 200	88 secs	200	1	0	0
TeraGen	10^6	19 secs	2	0	0	0
TeraSort	Output of TeraGen	26 secs	2	1	0	0
TeraValidate	Output of TeraSort	22 secs	1	0	0	0
Distributed Pentomino Solver	-	2hrs 39min 38 secs	2001	1	6	0
WordCount	1.2MB file	24 secs	2	1	0	0

2. Modified Hadoop 0.20

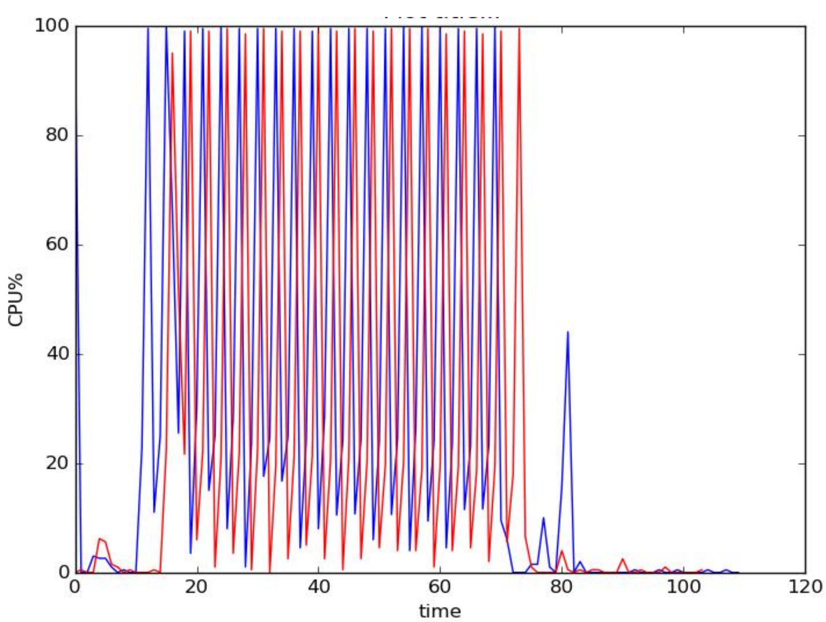
Job Name	Command Line arguments	Job Response time	No. of maps	No. of reduces	Speculative maps launched	Speculative reduces launched
PiEstimator	200, 200	78 secs	200	1	0	0
TeraGen	10^6	13 secs	2	0	0	1
TeraSort	Output of TeraGen	24 secs	2	1	0	0
TeraValidate	Output of TeraSort	21 secs	1	0	0	0
Distributed Pentomino Solver	-	2hrs 38min 45 secs	2001	1	0	1
WordCount	1.2 MB file	21 secs	2	1	0	0

CPU and Memory Stats

- Red line : Unmodified hadoop, and Blue line : Modified hadoop
    1. PiEstimator 200,200

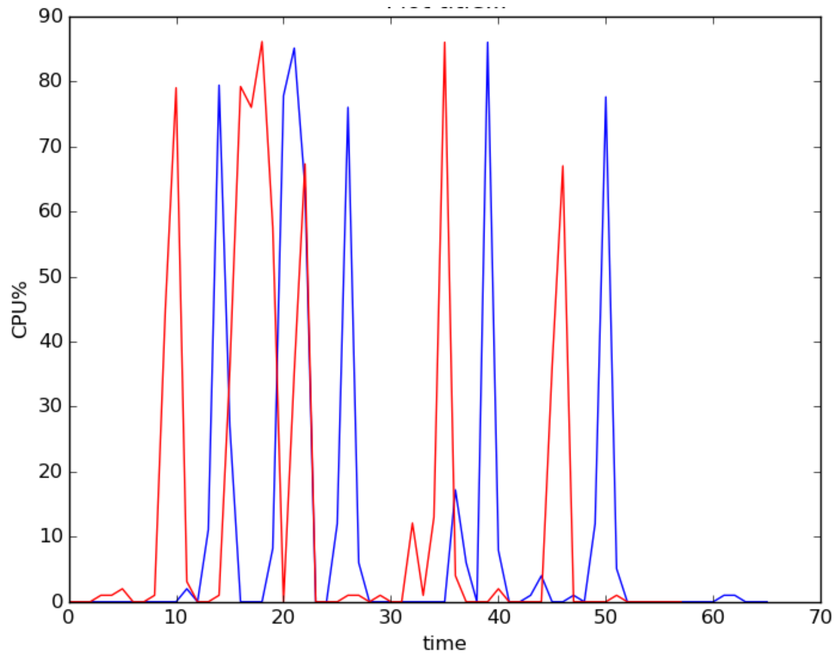


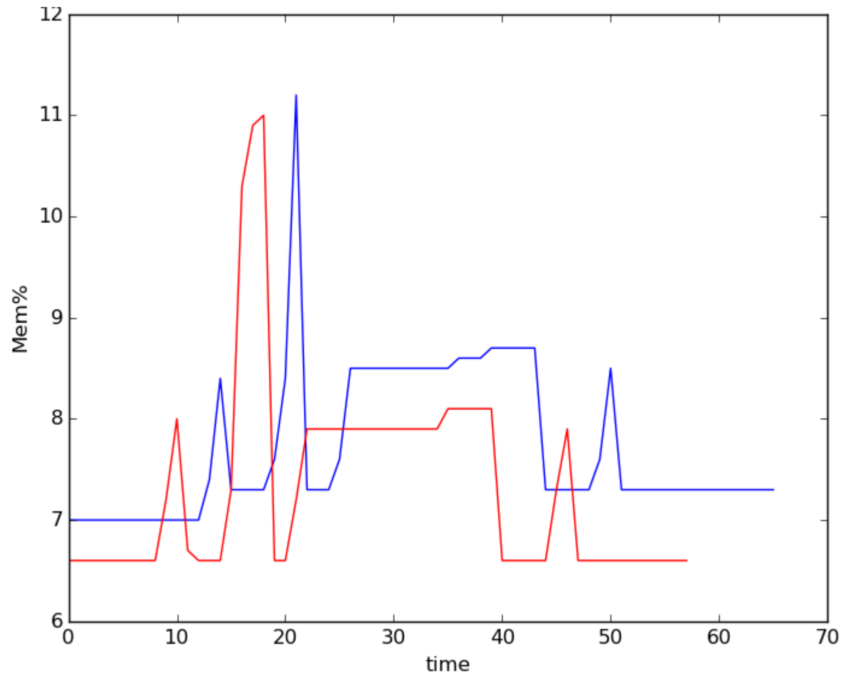
This graph explains the memory usage of a data node during the PiEstimator job. The spikes in the graph represent map or reduce tasks.



This graph explains the CPU usage of a data node during the PiEstimator job. The spikes in the graph represent map or reduce tasks.

2. WordCount 1.2MB text file





The above graphs represent CPU and memory usage of a data node with modified and unmodified hadoop versions for a Word Count job.

Conclusion

The modified hadoop 0.20 gives approximately the same job response times on different benchmarks run as shown in the previous section. The CPU and Memory usage for a few jobs are shown above. The CPU usage spikes are similar in both hadoop versions, whereas Memory usage seems to be slightly on the higher side in the modified hadoop framework.

We were able to observe a difference in number of speculative execution only in the dancingElephant(Distributed Pentomino Solver) job, where 6 maps were speculated in the existing hadoop version. Whereas, only 1 reduce task was speculated in the modified hadoop version. In the Pi Estimator job using Monte-Carlo simulation, with 200 maps and 200 inputs/map we observed that there was one speculative reduce task which started due to the slow progress of the reduce task. A reason for this might be that, Pi Estimator job has overlapping maps and reduce tasks.

A latent assumption here is that the tasks are linear in their progress. This is not a reliable assumption to make in general since the map, shuffle and reduce stages are pipelined and have spill-over mechanisms. Hence, calculating the progress for reduce by evenly weighting shuffle, sort and map is unreliable. To fix this, phase-specific metrics like completion time, mean and average number of spills are needed. Empirical evidence on large-scale cluster management systems, however, show that slightly complicated progress rate heuristics give a usable estimate.

Consider hadoop jobs where maps fetch data from external systems, and emit the data. The reducers in this are identity reducers. The data processed by these jobs is huge. There could be slow nodes in this cluster and some of the reducers run twice as slow as their counterparts. This could result in a long tail. Speculative execution would help greatly in such cases. However given the current hadoop, we have to select speculative execution for both maps and reducers. In this case hurting the map performance as they are fetching data from external systems thereby overloading the external systems.

Future Work

- Incorporate probability of node failure in the benefit calculation of launching a speculative task
- Calculate per-task resource utilization so that CPU, Memory, Disk and Network I/O utilization can incorporated into the cost calculation
- Come up with a better metric to calculate the estimated completion times of tasks

References

1) Zhenhua Guo, Geoffrey Fox, Mo Zhou, Yang Ruan. Improving Resource Utilization in MapReduce. {\it Indiana University Report}. May 2012.
2) Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, Ion Stoica. Improving MapReduce performance in heterogeneous environments. In {\it Proceedings of the 8th USENIX conference on Operating systems design and implementation}, p.29-42, December, 2008.
3) Apache Jira issues :-
    a. https://issues.apache.org/jira/browse/MAPREDUCE-3404
    b. https://issues.apache.org/jira/browse/HADOOP-1127
    c. https://issues.apache.org/jira/browse/HADOOP-2131
    d. https://issues.apache.org/jira/browse/MAPREDUCE-220

Appendix

Summary of changes in each java file

1. src/contrib/capacity-scheduler/src/java/org/apache/hadoop/mapred/CapacityTaskScheduler.java
   tip.hasSpeculativeTask(currentTime, progress) was replaced with tip.canBeSpeculated(currentTime) because this function was changed in TaskInProgress.java
2. src/contrib/capacity-scheduler/src/test/org/apache/hadoop/mapred/TestCapacityScheduler.java
   tip.hasSpeculativeTask(currentTime, progress) was replaced with tip.canBeSpeculated(currentTime) because this function was changed in TaskInProgress.java
3. src/contrib/fairscheduler/src/java/org/apache/hadoop/mapred/DefaultTaskSelector.java
   tip.hasSpeculativeTask(currentTime, progress) was replaced with tip.canBeSpeculated(currentTime) because this function was changed in TaskInProgress.java
4. src/mapred/mapred-default.xml
   Included variables SpeculativeCap, slowTaskThreshold and slowNodeThreshold for speculative cost calculation
5. src/mapred/org/apache/hadoop/mapred/JobInProgress.java
   a) Included the required variables and DataStatistics class for speculative cost calculation.
   b) Changed findSpeculativeTask() , to incorporate speculative task selection based on the metrics mentioned in the Algorithm
   c) Introduced getSpeculativeMap() and getSpeculativeReduce() to make calls to findSpeculativeTask() and get the task ID to be speculated.
   d) EstimatedTimeLeftComparator class introduced to compare the estimated completion time of two tasks, which is used to sort the candidates in findSpeculativeTask()
   e) isSlowTracker() – to determine if the TaskTracker node is too slow to run speculative tasks on

6. src/mapred/org/apache/hadoop/mapred/TaskInProgress.java
   a) Modified hasSpeculativeTask() to canBeSpeculated() so that an extra condition as mentioned in the algorithm is included to filter speculative tasks
   b) Calculate the bestProgressRate of a task in TaskTracker node with the help of getCurrentProgressRate()
7. src/mapred/org/apache/hadoop/mapred/JobConf.java
   Speculative map and reduce were enabled in this file. Ideally, speculative maps should be disabled and speculative reduces should be enabled. The reasons are stated in the Conclusion. Another good reason we found is that **multiple writes to HDFS is generally discouraged because slows down the write rates of other running map tasks**.

8. src/core/org/apache/Hadoop/util/LinuxResourceCalculatorPlugin.java
   Included this file at the last moment, to calculate per task resource utilization as mentioned by https://issues.apache.org/jira/browse/MAPREDUCE-220 . But wasn't able to use it correctly. Instantiating the class in JobInProgress.java and using its method gave figures which were similar to "**top"** output. In my honest opinion, if at all it calculates per task resource utilization it should have been associated with TaskInProgress class. However, figuring out its correct usage would be of paramount importance in calculating Speculative Cost as mentioned in Nidhi's proposal.

The following contains the changes in the Hadoop source code. Code portions colored in red were deleted and those in green were included. Few java files have no color in them in entirety; these files are new.

//Replaced tip.hasSpeculativeTask(currentTime, progress) with
//tip.canBeSpeculated(currentTime) in the following files :
// 1. src/contrib/capacity-
//scheduler/src/java/org/apache/hadoop/mapred/CapacityTaskScheduler.java
// 2. src/contrib/capacity-
//scheduler/src/test/org/apache/hadoop/mapred/TestCapacityScheduler.java
// 3. src/contrib/fairscheduler/src/java/org/apache/hadoop/mapred/DefaultTaskSelector.java

---

//Changed src/mapred/mapred-default.xml to include Speculative default variables.

```
<property>
  <name>mapred.speculative.execution.speculativeCap</name>
  <value>0.1</value>
  <description>The max percent (0-1) of running tasks that
  can be speculatively re-executed at any time.</description>
</property>


<property>
  <name>mapred.speculative.execution.slowTaskThreshold</name>
  <value>1.0</value>The number of standard deviations by which a task's
  ave progress-rates must be lower than the average of all running tasks'
  for the task to be considered too slow.
  <description>
  </description>
</property>

<property>
  <name>mapred.speculative.execution.slowNodeThreshold</name>
  <value>1.0</value>
  <description>The number of standard deviations by which a Task
  Tracker's ave map and reduce progress-rates (finishTime-dispatchTime)
  must be lower than the average of all successful map/reduce task's for
  the TT to be considered too slow to give a speculative task to.
  </description>
</property>

```

// Changed src/mapred/org/apache/hadoop/mapred/.java*

// 1. JobInProgress.java

```java
import java.util.Collections;
import java.util.Comparator;
import java.util.HashSet;
import java.util.HashMap;
```

```java
  // Don't lower speculativeCap below one TT's worth (for small clusters)
  private static final int MIN_SPEC_CAP = 10;

  private static final float MIN_SLOTS_CAP = 0.01f;
```

```java
  //thresholds for speculative execution
  private float slowTaskThreshold;
  private float speculativeCap;
  private float slowNodeThreshold; //standard deviations
```

```java
  //Statistics are maintained for a couple of things
  //mapTaskStats is used for maintaining statistics about
  //the completion time of map tasks on the trackers. On a per
  //tracker basis, the mean time for task completion is maintained
  private DataStatistics mapTaskStats = new DataStatistics();
  //reduceTaskStats is used for maintaining statistics about
  //the completion time of reduce tasks on the trackers. On a per
  //tracker basis, the mean time for task completion is maintained
  private DataStatistics reduceTaskStats = new DataStatistics();
  //trackerMapStats used to maintain a mapping from the tracker to the
  //the statistics about completion time of map tasks
  private Map<String,DataStatistics> trackerMapStats =
    new HashMap<String,DataStatistics>();
  //trackerReduceStats used to maintain a mapping from the tracker to the
  //the statistics about completion time of reduce tasks
  private Map<String,DataStatistics> trackerReduceStats =
    new HashMap<String,DataStatistics>();
  //runningMapStats used to maintain the RUNNING map tasks' statistics
  private DataStatistics runningMapTaskStats = new DataStatistics();
  //runningReduceStats used to maintain the RUNNING reduce tasks' statistics
  private DataStatistics runningReduceTaskStats = new DataStatistics();
```

```java
    this.jobtracker = null;
    this.jobtracker = tracker;
```

```
    hasSpeculativeMaps = conf.getMapSpeculativeExecution();
    hasSpeculativeReduces = conf.getReduceSpeculativeExecution();
    this.nonLocalMaps = new LinkedList<TaskInProgress>();
    this.nonLocalRunningMaps = new LinkedHashSet<TaskInProgress>();
    this.runningMapCache = new IdentityHashMap<Node, Set<TaskInProgress>>();
    this.nonRunningReduces = new LinkedList<TaskInProgress>();
    this.runningReduces = new LinkedHashSet<TaskInProgress>();
    this.resourceEstimator = new ResourceEstimator(this);
    this.status = new JobStatus(jobid, 0.0f, 0.0f, JobStatus.PREP);
    this.taskCompletionEvents = new ArrayList<TaskCompletionEvent>
    (numMapTasks + numReduceTasks + 10);

    this.slowTaskThreshold = Math.max(0.0f,
        conf.getFloat("mapred.speculative.execution.slowTaskThreshold",1.0f));
    this.speculativeCap = conf.getFloat(
        "mapred.speculative.execution.speculativeCap",0.1f);
    this.slowNodeThreshold = conf.getFloat(
        "mapred.speculative.execution.slowNodeThreshold",1.0f);
```

```

    this.nonLocalMaps = new LinkedList<TaskInProgress>();
    this.nonLocalRunningMaps = new LinkedHashSet<TaskInProgress>();
    this.runningMapCache = new IdentityHashMap<Node, Set<TaskInProgress>>();
    this.nonRunningReduces = new LinkedList<TaskInProgress>();
    this.runningReduces = new LinkedHashSet<TaskInProgress>();
    this.slowTaskThreshold = Math.max(0.0f,
        conf.getFloat("mapred.speculative.execution.slowTaskThreshold",1.0f));
    this.speculativeCap = conf.getFloat(
        "mapred.speculative.execution.speculativeCap",0.1f);
    this.slowNodeThreshold = conf.getFloat(
        "mapred.speculative.execution.slowNodeThreshold",1.0f);

```

```
    int target = findNewMapTask(tts, clusterSize, numUniqueHosts, anyCacheLevel,
                                status.mapProgress());
```

```
    int target = findNewMapTask(tts, clusterSize, numUniqueHosts,
        anyCacheLevel);
```

```
    int target = findNewMapTask(tts, clusterSize, numUniqueHosts, maxLevel,
```

```
1005                                         status.mapProgress());
```

```
1071      int target = findNewMapTask(tts, clusterSize, numUniqueHosts, maxLevel);
```

```
1028                                         NON_LOCAL_CACHE_LEVEL, status.mapProgress());
1094                                         NON_LOCAL_CACHE_LEVEL);
```

```
1206      int  target = findNewReduceTask(tts, clusterSize, numUniqueHosts,
1207                                           status.reduceProgress());
```

```
1272      int  target = findNewReduceTask(tts, clusterSize, numUniqueHosts);
```

```
1270        if (tip.getActiveTasks().size() > 1)
1335        if (tip.isSpeculating()) {
1337          LOG.debug("Chosen speculative task, current speculativeMap task count: "
1338              + speculativeMapTasks);
1339        }
```

```
1277        if (tip.getActiveTasks().size() > 1)
1345        if (tip.isSpeculating()) {
```

```
1347          LOG.debug("Chosen speculative task, current speculativeReduce task count: "
1348            + speculativeReduceTasks);
1349        }
```

```
1688    public boolean hasSpeculativeMaps() {
1689      return hasSpeculativeMaps;
1690    }
1691
1692    public boolean hasSpeculativeReduces() {
1693      return hasSpeculativeReduces;
1694    }
1695
```

```
1616     * Find a speculative task
1617     * @param list a list of tips
1618     * @param taskTracker the tracker that has requested a tip
1619     * @param avgProgress the average progress for speculation
1620     * @param currentTime current time in milliseconds
1621     * @param shouldRemove whether to remove the tips
1622     * @return a tip that can be speculated on the tracker
```

```
   * Retrieve a task for speculation.
   * If a task slot becomes available and there are less than SpeculativeCap
   * speculative tasks running:
   *  1)Ignore the request if the TT's progressRate is < SlowNodeThreshold
   *  2)Choose candidate tasks - those tasks whose progress rate is below
   *    slowTaskThreshold * mean(progress-rates)
   *  3)Speculate task that's expected to complete last
   * @param list pool of tasks to choose from
   * @param taskTrackerName the name of the TaskTracker asking for a task
   * @param taskTrackerHost the hostname of the TaskTracker asking for a task
   * @return the TIP to speculatively re-execute
```

```
  private synchronized TaskInProgress findSpeculativeTask(
      Collection<TaskInProgress> list, TaskTrackerStatus ttStatus,
      double avgProgress, long currentTime, boolean shouldRemove) {
```

```
  protected synchronized TaskInProgress findSpeculativeTask(
      Collection<TaskInProgress> list, String taskTrackerName,
      String taskTrackerHost) {
    if (list.isEmpty()) {
      return null;
    }
    long now = jobtracker.getClock().getTime();
    if (isSlowTracker(taskTrackerName) || atSpeculativeCap(list)) {
      return null;
    }
    // List of speculatable candidates, start with all, and chop it down
    ArrayList<TaskInProgress> candidates = new ArrayList<TaskInProgress>(list);
```

```
    Iterator<TaskInProgress> iter = list.iterator();
```

```
    Iterator<TaskInProgress> iter = candidates.iterator();
```

```
      // should never be true! (since we delete completed/failed tasks)
      if (!tip.isRunning()) {
```

```
      if (tip.hasRunOnMachine(taskTrackerHost, taskTrackerName) ||
          !tip.canBeSpeculated(now)) {
        //remove it from candidates
```

```

      if (!tip.hasRunOnMachine(ttStatus.getHost(),
                               ttStatus.getTrackerName())) {
        if (tip.hasSpeculativeTask(currentTime, avgProgress)) {
          // In case of shared list we don't remove it. Since the TIP failed
          // on this tracker can be scheduled on some other tracker.
          if (shouldRemove) {
            iter.remove(); //this tracker is never going to run it again
```

```
          return tip;
    //resort according to expected time till completion
    Comparator<TaskInProgress> LateComparator =
      new EstimatedTimeLeftComparator(now);
    Collections.sort(candidates, LateComparator);
    if (candidates.size() > 0 ) {
      TaskInProgress tip = candidates.get(0);
      if (LOG.isDebugEnabled()) {
        LOG.debug("Chose task " + tip.getTIPId() + ". Statistics: Task's : " +
            tip.getCurrentProgressRate(now) + " Job's : " +
            (tip.isMapTask() ? runningMapTaskStats : runningReduceTaskStats));
      }
    return tip;
```

```
  private synchronized TaskInProgress getSpeculativeReduce(
      String taskTrackerName, String taskTrackerHost) {
    TaskInProgress tip = findSpeculativeTask(
        runningReduces, taskTrackerName, taskTrackerHost);
    if (tip != null) {
      LOG.info("Choosing reduce task " + tip.getTIPId() +
          " for speculative execution");
    }else {
      LOG.debug("No speculative map task found for tracker " + taskTrackerHost);
    }
    return tip;
  }

    /**
     * Check to see if the maximum number of speculative tasks are
     * already being executed currently.
     * @param tasks the set of tasks to test
```

```java
   * @return has the cap been reached?
   */
  private boolean atSpeculativeCap(Collection<TaskInProgress> tasks) {
    float numTasks = tasks.size();
    if (numTasks == 0){
      return true; // avoid divide by zero
    }

    //return true if totalSpecTask < max(10, 0.01 * total-slots,
    //                                  0.1 * total-running-tasks)

    if (speculativeMapTasks + speculativeReduceTasks < MIN_SPEC_CAP) {
      return false; // at least one slow tracker's worth of slots(default=10)
    }
    ClusterStatus c = jobtracker.getClusterStatus(false);
    int numSlots = c.getMaxMapTasks() + c.getMaxReduceTasks();
    if ((float)(speculativeMapTasks + speculativeReduceTasks) <
      numSlots * MIN_SLOTS_CAP) {
      return false;
    }
    boolean atCap = (((float)(speculativeMapTasks+
        speculativeReduceTasks)/numTasks) >= speculativeCap);
    if (LOG.isDebugEnabled()) {
      LOG.debug("SpeculativeCap is "+speculativeCap+", specTasks/numTasks is " +
          ((float)(speculativeMapTasks+speculativeReduceTasks)/numTasks)+
          ", so atSpecCap() is returning "+atCap);
    }
    return atCap;
  }

  /**
   * A class for comparing the estimated time to completion of two tasks
   */
  private static class EstimatedTimeLeftComparator
  implements Comparator<TaskInProgress> {
    private long time;
    public EstimatedTimeLeftComparator(long now) {
      this.time = now;
    }
    /**
     * Estimated time to completion is measured as:
     *   % of task left to complete (1 - progress) / progress rate of the task.
     *
```

```java
     * This assumes that tasks are linear in their progress, which is
     * often wrong, especially since progress for reducers is currently
     * calculated by evenly weighting their three stages (shuffle, sort, map)
     * which rarely account for 1/3 each. This should be fixed in the future
     * by calculating progressRate more intelligently or splitting these
     * multi-phase tasks into individual tasks.
     *
     * The ordering this comparator defines is: task1 < task2 if task1 is
     * estimated to finish farther in the future => compare(t1,t2) returns -1
     */
    public int compare(TaskInProgress tip1, TaskInProgress tip2) {
      //we have to use the Math.max in the denominator to avoid divide by zero
      //error because prog and progRate can both be zero (if one is zero,
      //the other one will be 0 too).
      //We use inverse of time_reminaing=[(1- prog) / progRate]
      //so that (1-prog) is in denom. because tasks can have arbitrarily
      //low progRates in practice (e.g. a task that is half done after 1000
      //seconds will have progRate of 0.0000005) so we would rather
      //use Math.maxnon (1-prog) by putting it in the denominator
      //which will cause tasks with prog=1 look 99.99% done instead of 100%
      //which is okay
      double t1 = tip1.getCurrentProgressRate(time) / Math.max(0.0001,
          1.0 - tip1.getProgress());
      double t2 = tip2.getCurrentProgressRate(time) / Math.max(0.0001,
          1.0 - tip2.getProgress());
      if (t1 < t2) return -1;
      else if (t2 < t1) return 1;
      else return 0;
    }
  }

  /**
   * Compares the ave progressRate of tasks that have finished on this
   * taskTracker to the ave of all succesfull tasks thus far to see if this
   * TT one is too slow for speculating.
   * slowNodeThreshold is used to determine the number of standard deviations
   * @param taskTracker the name of the TaskTracker we are checking
   * @return is this TaskTracker slow
   */
  protected boolean isSlowTracker(String taskTracker) {
    if (trackerMapStats.get(taskTracker) != null &&
        trackerMapStats.get(taskTracker).mean() -
        mapTaskStats.mean() > mapTaskStats.std()*slowNodeThreshold) {
```

```
      if (LOG.isDebugEnabled()) {
        LOG.debug("Tracker " + taskTracker +
            " declared slow. trackerMapStats.get(taskTracker).mean() :" + trackerMapSta
            " mapTaskStats :" + mapTaskStats);
      }
      return true;
    }
    if (trackerReduceStats.get(taskTracker) != null &&
        trackerReduceStats.get(taskTracker).mean() -
        reduceTaskStats.mean() > reduceTaskStats.std()*slowNodeThreshold) {
      if (LOG.isDebugEnabled()) {
        LOG.debug("Tracker " + taskTracker +
            " declared slow. trackerReduceStats.get(taskTracker).mean() :" +
trackerReduceStats.get(taskTracker).mean() +
            " reduceTaskStats :" + reduceTaskStats);
      }
      return true;
    }
    return false;
  }

  static class DataStatistics{
    private int count = 0;
    private double sum = 0;
    private double sumSquares = 0;

    public DataStatistics() {
    }

    public DataStatistics(double initNum) {
      this.count = 1;
      this.sum = initNum;
      this.sumSquares = initNum * initNum;
    }

    public void add(double newNum) {
      this.count++;
      this.sum += newNum;
      this.sumSquares += newNum * newNum;
    }

    public void updateStatistics(double old, double update) {
      sub(old);
```

```java
      add(update);
    }
    private void sub(double oldNum) {
      this.count--;
      this.sum -= oldNum;
      this.sumSquares -= oldNum * oldNum;
    }

    public double mean() {
      return sum/count;
    }

    public double var() {
      // E(X^2) - E(X)^2
      return (sumSquares/count) - mean() * mean();
    }

    public double std() {
      return Math.sqrt(this.var());
    }

    public String toString() {
      return "DataStatistics: count is " + count + ", sum is " + sum +
      ", sumSquares is " + sumSquares + " mean is " + mean() + " std() is " + std();
    }
  private void updateTaskTrackerStats(TaskInProgress tip, TaskTrackerStatus ttStatus,
      Map<String,DataStatistics> trackerStats, DataStatistics overallStats) {
    float tipDuration = tip.getExecFinishTime()-tip.getDispatchTime();
    DataStatistics ttStats =
      trackerStats.get(ttStatus.getTrackerName());
    double oldMean = 0.0d;
    //We maintain the mean of TaskTrackers' means. That way, we get a single
    //data-point for every tracker (used in the evaluation in isSlowTracker)
    if (ttStats != null) {
      oldMean = ttStats.mean();
      ttStats.add(tipDuration);
      overallStats.updateStatistics(oldMean, ttStats.mean());
    } else {
      trackerStats.put(ttStatus.getTrackerName(),
          (ttStats = new DataStatistics(tipDuration)));
      overallStats.add(tipDuration);
    }
    if (LOG.isDebugEnabled()) {
```

```
      LOG.debug("Added mean of " +ttStats.mean() + " to trackerStats of type "+
          (tip.isMapTask() ? "Map" : "Reduce") +
          " on "+ttStatus.getTrackerName()+". DataStatistics is now: " +
          trackerStats.get(ttStatus.getTrackerName()));
    }
  }

  public void updateStatistics(double oldProg, double newProg, boolean isMap) {
    if (isMap) {
      runningMapTaskStats.updateStatistics(oldProg, newProg);
    } else {
      runningReduceTaskStats.updateStatistics(oldProg, newProg);
    }
  }

  public DataStatistics getRunningTaskStatistics(boolean isMap) {
    if (isMap) {
      return runningMapTaskStats;
    } else {
      return runningReduceTaskStats;
    }
  }

  public float getSlowTaskThreshold() {
    return slowTaskThreshold;
  }


  private void decrementSpeculativeCount(boolean wasSpeculating,
      TaskInProgress tip) {
    if (wasSpeculating) {
      if (tip.isMapTask()) {
        speculativeMapTasks--;
        LOG.debug("Decrement count. Current speculativeMap task count: " +
            speculativeMapTasks);
      } else {
        speculativeReduceTasks--;
        LOG.debug("Decremented count. Current speculativeReduce task count: " +
            speculativeReduceTasks);
      }
    }
  }

```

```
    boolean wasSpeculating = tip.isSpeculating();

    decrementSpeculativeCount(wasSpeculating, tip);
```

//2. TaskInProgress.java

```java
import org.apache.hadoop.mapred.JobInProgress.DataStatistics;
```

```java
  private double oldProgressRate;
```

```java
  private long startTime = 0;
  private long execStartTime = 0;
```

```java
  private long dispatchTime = 0;    // most recent time task attempt given to TT
  private long execStartTime = 0;   // when we started first task-attempt
```

```java
  public long getStartTime() {
    return startTime;
```

```java
  public long getDispatchTime(){
    return this.dispatchTime;
```

```java
    // if task is a cleanup attempt, do not replace the complete status,
    // update only specific fields.
    // For example, startTime should not be updated,
    // but finishTime has to be updated.
    if (!isCleanupAttempt(taskid)) {
      taskStatuses.put(taskid, status);
      if ((isMapTask() && job.hasSpeculativeMaps()) ||
          (!isMapTask() && job.hasSpeculativeReduces())) {
        long now = jobtracker.getClock().getTime();
        double oldProgRate = getOldProgressRate();
        double currProgRate = getCurrentProgressRate(now);
        job.updateStatistics(oldProgRate, currProgRate, isMapTask());
        //we need to store the current progress rate, so that we can
        //update statistics accurately the next time we invoke
        //updateStatistics
        setProgressRate(currProgRate);
      }
```

```java
   * Return whether the TIP has a speculative task to run.  We
   * only launch a speculative task if the current TIP is really
   * far behind, and has been behind for a non-trivial amount of
```

```
   * time.
```

```
   * Can this task be speculated? This requires that it isn't done or almost
   * done and that it isn't already being speculatively executed.
   *
```

```
  boolean hasSpeculativeTask(long currentTime, double averageProgress) {
    //
    // REMIND - mjc - these constants should be examined
    // in more depth eventually...
    //

    if (!skipping && activeTasks.size() <= MAX_TASK_EXECS &&
        (averageProgress - progress >= SPECULATIVE_GAP) &&
        (currentTime - startTime >= SPECULATIVE_LAG)
        && completes == 0 && !isOnlyCommitPending()) {
      return true;
```

```
  boolean canBeSpeculated(long currentTime) {
    DataStatistics taskStats = job.getRunningTaskStatistics(isMapTask());
    if (LOG.isDebugEnabled()) {
      LOG.debug("activeTasks.size(): " + activeTasks.size() + " "
          + activeTasks.firstKey() + " task's progressrate: " +
          getCurrentProgressRate(currentTime) +
          " taskStats : " + taskStats);
```

```
    return false;
    return (!skipping && isRunnable() && isRunning() &&
        activeTasks.size() <= MAX_TASK_EXECS &&
        currentTime - dispatchTime >= SPECULATIVE_LAG &&
        completes == 0 && !isOnlyCommitPending() &&
        (taskStats.mean() - getCurrentProgressRate(currentTime) >
              taskStats.std() * job.getSlowTaskThreshold()));
  }

  boolean isSpeculating() {
   return (activeTasks.size() > MAX_TASK_EXECS);

   * Compare most recent task attempts dispatch time to current system time so
   * that task progress rate will slow down as time proceeds even if no progress
   * is reported for the task. This allows speculative tasks to be launched for
```

```
   * tasks on slow/dead TT's before we realize the TT is dead/slow. Skew isn't
   * an issue since both times are from the JobTrackers perspective.
   * @return the progress rate from the active task that is doing best
   */
  public double getCurrentProgressRate(long currentTime) {
    double bestProgressRate = 0;
    for (TaskStatus ts : taskStatuses.values()){
      double progressRate = ts.getProgress()/Math.max(1,
          currentTime - dispatchTime);
      if ((ts.getRunState() == TaskStatus.State.RUNNING  ||
          ts.getRunState() == TaskStatus.State.SUCCEEDED) &&
          progressRate > bestProgressRate){
        bestProgressRate = progressRate;
      }
    }
    return bestProgressRate;
  }

  private void setProgressRate(double rate) {
    oldProgressRate = rate;
  }
  private double getOldProgressRate() {
    return oldProgressRate;
  }
```

//3. JobConf.java

```
  public void setMapSpeculativeExecution(boolean speculativeExecution) {
    //setBoolean("mapred.map.tasks.speculative.execution", speculativeExecution);
    setBoolean("mapred.map.tasks.speculative.execution", true);//Changed by Kartik
  }

public void setReduceSpeculativeExecution(boolean speculativeExecution) {
    //setBoolean("mapred.reduce.tasks.speculative.execution",
    //          speculativeExecution);
    setBoolean("mapred.reduce.tasks.speculative.execution",
             true);//Changed by Kartik
  }
```

*//Entirely new file – src/core/org/apache/Hadoop/util/**
// 1. LinuxResourceCalculatorPlugin.java

```
package org.apache.hadoop.util;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * Plugin to calculate resource information on Linux systems.
 */
public class LinuxResourceCalculatorPlugin extends ResourceCalculatorPlugin {
  private static final Log LOG =
      LogFactory.getLog(LinuxResourceCalculatorPlugin.class);

  /**
   * proc's meminfo virtual file has keys-values in the format
   * "key:[ \t]*value[ \t]kB".
   */
  private static final String PROCFS_MEMFILE = "/proc/meminfo";
  private static final Pattern PROCFS_MEMFILE_FORMAT =
      Pattern.compile("^([a-zA-Z]*):[ \t]*([0-9]*)[ \t]kB");

  // We need the values for the following keys in meminfo
  private static final String MEMTOTAL_STRING = "MemTotal";
  private static final String SWAPTOTAL_STRING = "SwapTotal";
  private static final String MEMFREE_STRING = "MemFree";
  private static final String SWAPFREE_STRING = "SwapFree";
  private static final String INACTIVE_STRING = "Inactive";
  private static final int UNAVAILABLE = -1;

  /**
   * Patterns for parsing /proc/cpuinfo
   */
  private static final String PROCFS_CPUINFO = "/proc/cpuinfo";
  private static final Pattern PROCESSOR_FORMAT =
```

```
    Pattern.compile("^processor[ \t]:[ \t]*([0-9]*)");
private static final Pattern FREQUENCY_FORMAT =
    Pattern.compile("^cpu MHz[ \t]*:[ \t]*([0-9.]*)");

/**
 * Pattern for parsing /proc/stat
 */
private static final String PROCFS_STAT = "/proc/stat";
private static final Pattern CPU_TIME_FORMAT =
  Pattern.compile("^cpu[ \t]*([0-9]*)" +
                               "[ \t]*([0-9]*)[ \t]*([0-9]*)[ \t].*");

private String procfsMemFile;
private String procfsCpuFile;
private String procfsStatFile;
long jiffyLengthInMillis;

private long ramSize = 0;
private long swapSize = 0;
private long ramSizeFree = 0;  // free ram space on the machine (kB)
private long swapSizeFree = 0; // free swap space on the machine (kB)
private long inactiveSize = 0; // inactive cache memory (kB)
private int numProcessors = 0; // number of processors on the system
private long cpuFrequency = 0L; // CPU frequency on the system (kHz)
private long cumulativeCpuTime = 0L; // CPU used time since system is on (ms)
private long lastCumulativeCpuTime = 0L; // CPU used time read last time (ms)
// Unix timestamp while reading the CPU time (ms)
private float cpuUsage = UNAVAILABLE;
private long sampleTime = UNAVAILABLE;
private long lastSampleTime = UNAVAILABLE;
private ProcfsBasedProcessTree pTree = null;

boolean readMemInfoFile = false;
boolean readCpuInfoFile = false;

/**
 * Get current time
 * @return Unix time stamp in millisecond
 */
long getCurrentTime() {
  return System.currentTimeMillis();
}

public LinuxResourceCalculatorPlugin() {
  procfsMemFile = PROCFS_MEMFILE;
  procfsCpuFile = PROCFS_CPUINFO;
```

```
    procfsStatFile = PROCFS_STAT;
    jiffyLengthInMillis = ProcfsBasedProcessTree.JIFFY_LENGTH_IN_MILLIS;
    String pid = System.getenv().get("JVM_PID");
    pTree = new ProcfsBasedProcessTree(pid);
  }

  /**
   * Constructor which allows assigning the /proc/ directories. This will be
   * used only in unit tests
   * @param procfsMemFile fake file for /proc/meminfo
   * @param procfsCpuFile fake file for /proc/cpuinfo
   * @param procfsStatFile fake file for /proc/stat
   * @param jiffyLengthInMillis fake jiffy length value
   */
  public LinuxResourceCalculatorPlugin(String procfsMemFile,
                                       String procfsCpuFile,
                                       String procfsStatFile,
                                       long jiffyLengthInMillis) {
    this.procfsMemFile = procfsMemFile;
    this.procfsCpuFile = procfsCpuFile;
    this.procfsStatFile = procfsStatFile;
    this.jiffyLengthInMillis = jiffyLengthInMillis;
    String pid = System.getenv().get("JVM_PID");
    pTree = new ProcfsBasedProcessTree(pid);
  }

  /**
   * Read /proc/meminfo, parse and compute memory information only once
   */
  private void readProcMemInfoFile() {
    readProcMemInfoFile(false);
  }

  /**
   * Read /proc/meminfo, parse and compute memory information
   * @param readAgain if false, read only on the first time
   */
  private void readProcMemInfoFile(boolean readAgain) {

    if (readMemInfoFile && !readAgain) {
      return;
    }

    // Read "/proc/memInfo" file
    BufferedReader in = null;
    FileReader fReader = null;
```

```
try {
  fReader = new FileReader(procfsMemFile);
  in = new BufferedReader(fReader);
} catch (FileNotFoundException f) {
  // shouldn't happen....
  return;
}

Matcher mat = null;

try {
  String str = in.readLine();
  while (str != null) {
    mat = PROCFS_MEMFILE_FORMAT.matcher(str);
    if (mat.find()) {
      if (mat.group(1).equals(MEMTOTAL_STRING)) {
        ramSize = Long.parseLong(mat.group(2));
      } else if (mat.group(1).equals(SWAPTOTAL_STRING)) {
        swapSize = Long.parseLong(mat.group(2));
      } else if (mat.group(1).equals(MEMFREE_STRING)) {
        ramSizeFree = Long.parseLong(mat.group(2));
      } else if (mat.group(1).equals(SWAPFREE_STRING)) {
        swapSizeFree = Long.parseLong(mat.group(2));
      } else if (mat.group(1).equals(INACTIVE_STRING)) {
        inactiveSize = Long.parseLong(mat.group(2));
      }
    }
    str = in.readLine();
  }
} catch (IOException io) {
  LOG.warn("Error reading the stream " + io);
} finally {
  // Close the streams
  try {
    fReader.close();
    try {
      in.close();
    } catch (IOException i) {
      LOG.warn("Error closing the stream " + in);
    }
  } catch (IOException i) {
    LOG.warn("Error closing the stream " + fReader);
  }
}

readMemInfoFile = true;
```

```
}

/**
 * Read /proc/cpuinfo, parse and calculate CPU information
 */
private void readProcCpuInfoFile() {
  // This directory needs to be read only once
  if (readCpuInfoFile) {
    return;
  }
  // Read "/proc/cpuinfo" file
  BufferedReader in = null;
  FileReader fReader = null;
  try {
    fReader = new FileReader(procfsCpuFile);
    in = new BufferedReader(fReader);
  } catch (FileNotFoundException f) {
    // shouldn't happen....
    return;
  }
  Matcher mat = null;
  try {
    numProcessors = 0;
    String str = in.readLine();
    while (str != null) {
      mat = PROCESSOR_FORMAT.matcher(str);
      if (mat.find()) {
        numProcessors++;
      }
      mat = FREQUENCY_FORMAT.matcher(str);
      if (mat.find()) {
        cpuFrequency = (long)(Double.parseDouble(mat.group(1)) * 1000); // kHz
      }
      str = in.readLine();
    }
  } catch (IOException io) {
    LOG.warn("Error reading the stream " + io);
  } finally {
    // Close the streams
    try {
      fReader.close();
      try {
        in.close();
      } catch (IOException i) {
        LOG.warn("Error closing the stream " + in);
      }
```

```
      } catch (IOException i) {
        LOG.warn("Error closing the stream " + fReader);
      }
    }
    readCpuInfoFile = true;
  }

  /**
   * Read /proc/stat file, parse and calculate cumulative CPU
   */
  private void readProcStatFile() {
    // Read "/proc/stat" file
    BufferedReader in = null;
    FileReader fReader = null;
    try {
      fReader = new FileReader(procfsStatFile);
      in = new BufferedReader(fReader);
    } catch (FileNotFoundException f) {
      // shouldn't happen....
      return;
    }

    Matcher mat = null;
    try {
      String str = in.readLine();
      while (str != null) {
        mat = CPU_TIME_FORMAT.matcher(str);
        if (mat.find()) {
          long uTime = Long.parseLong(mat.group(1));
          long nTime = Long.parseLong(mat.group(2));
          long sTime = Long.parseLong(mat.group(3));
          cumulativeCpuTime = uTime + nTime + sTime; // milliseconds
          break;
        }
        str = in.readLine();
      }
      cumulativeCpuTime *= jiffyLengthInMillis;
    } catch (IOException io) {
      LOG.warn("Error reading the stream " + io);
    } finally {
      // Close the streams
      try {
        fReader.close();
        try {
          in.close();
        } catch (IOException i) {
```

```
        LOG.warn("Error closing the stream " + in);
      }
    } catch (IOException i) {
      LOG.warn("Error closing the stream " + fReader);
    }
  }
}

/** {@inheritDoc} */
@Override
public long getPhysicalMemorySize() {
  readProcMemInfoFile();
  return ramSize * 1024;
}

/** {@inheritDoc} */
@Override
public long getVirtualMemorySize() {
  readProcMemInfoFile();
  return (ramSize + swapSize) * 1024;
}

/** {@inheritDoc} */
@Override
public long getAvailablePhysicalMemorySize() {
  readProcMemInfoFile(true);
  return (ramSizeFree + inactiveSize) * 1024;
}

/** {@inheritDoc} */
@Override
public long getAvailableVirtualMemorySize() {
  readProcMemInfoFile(true);
  return (ramSizeFree + swapSizeFree + inactiveSize) * 1024;
}

/** {@inheritDoc} */
@Override
public int getNumProcessors() {
  readProcCpuInfoFile();
  return numProcessors;
}

/** {@inheritDoc} */
@Override
public long getCpuFrequency() {
```

```
  readProcCpuInfoFile();
  return cpuFrequency;
}

/** {@inheritDoc} */
@Override
public long getCumulativeCpuTime() {
  readProcStatFile();
  return cumulativeCpuTime;
}

/** {@inheritDoc} */
@Override
public float getCpuUsage() {
  readProcStatFile();
  sampleTime = getCurrentTime();
  if (lastSampleTime == UNAVAILABLE ||
      lastSampleTime > sampleTime) {
    // lastSampleTime > sampleTime may happen when the system time is changed
    lastSampleTime = sampleTime;
    lastCumulativeCpuTime = cumulativeCpuTime;
    return cpuUsage;
  }
  // When lastSampleTime is sufficiently old, update cpuUsage.
  // Also take a sample of the current time and cumulative CPU time for the
  // use of the next calculation.
  final long MINIMUM_UPDATE_INTERVAL = 10 * jiffyLengthInMillis;
  if (sampleTime > lastSampleTime + MINIMUM_UPDATE_INTERVAL) {
            cpuUsage = (float)(cumulativeCpuTime - lastCumulativeCpuTime) * 100F /
                       ((float)(sampleTime - lastSampleTime) * getNumProcessors());
            lastSampleTime = sampleTime;
    lastCumulativeCpuTime = cumulativeCpuTime;
  }
  return cpuUsage;
}

/**
 * Test the {@link LinuxResourceCalculatorPlugin}
 *
 * @param args
 */
public static void main(String[] args) {
  LinuxResourceCalculatorPlugin plugin = new LinuxResourceCalculatorPlugin();
  System.out.println("Physical memory Size (bytes) : "
      + plugin.getPhysicalMemorySize());
  System.out.println("Total Virtual memory Size (bytes) : "
```

```
        + plugin.getVirtualMemorySize());
    System.out.println("Available Physical memory Size (bytes) : "
        + plugin.getAvailablePhysicalMemorySize());
    System.out.println("Total Available Virtual memory Size (bytes) : "
        + plugin.getAvailableVirtualMemorySize());
    System.out.println("Number of Processors : " + plugin.getNumProcessors());
    System.out.println("CPU frequency (kHz) : " + plugin.getCpuFrequency());
    System.out.println("Cumulative CPU time (ms) : " +
            plugin.getCumulativeCpuTime());
    try {
      // Sleep so we can compute the CPU usage
      Thread.sleep(500L);
    } catch (InterruptedException e) {
      // do nothing
    }
    System.out.println("CPU usage % : " + plugin.getCpuUsage());
  }

  @Override
  public ProcResourceValues getProcResourceValues() {
    pTree = pTree.getProcessTree();
    long cpuTime = pTree.getCumulativeCpuTime();
    long pMem = pTree.getCumulativeRssmem();
    long vMem = pTree.getCumulativeVmem();
    return new ProcResourceValues(cpuTime, pMem, vMem);
  }
}
```

// 2. ResourceCalculatorPlugin.java

```
package org.apache.hadoop.util;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.util.ReflectionUtils;

/**
 * Plugin to calculate resource information on the system.
 *
 */
public abstract class ResourceCalculatorPlugin extends Configured {

  /**
   * Obtain the total size of the virtual memory present in the system.
   *
   * @return virtual memory size in bytes.
   */
  public abstract long getVirtualMemorySize();

  /**
   * Obtain the total size of the physical memory present in the system.
   *
   * @return physical memory size bytes.
   */
  public abstract long getPhysicalMemorySize();

  /**
   * Obtain the total size of the available virtual memory present
   * in the system.
   *
   * @return available virtual memory size in bytes.
   */
  public abstract long getAvailableVirtualMemorySize();

  /**
   * Obtain the total size of the available physical memory present
   * in the system.
   *
   * @return available physical memory size bytes.
   */
  public abstract long getAvailablePhysicalMemorySize();
```

```
/**
 * Obtain the total number of processors present on the system.
 *
 * @return number of processors
 */
public abstract int getNumProcessors();

/**
 * Obtain the CPU frequency of on the system.
 *
 * @return CPU frequency in kHz
 */
public abstract long getCpuFrequency();

/**
 * Obtain the cumulative CPU time since the system is on.
 *
 * @return cumulative CPU time in milliseconds
 */
public abstract long getCumulativeCpuTime();

/**
 * Obtain the CPU usage % of the machine. Return -1 if it is unavailable
 *
 * @return CPU usage in %
 */
public abstract float getCpuUsage();

/**
 * Obtain resource status used by current process tree.
 */
public abstract ProcResourceValues getProcResourceValues();

public static class ProcResourceValues {
  private final long cumulativeCpuTime;
  private final long physicalMemorySize;
  private final long virtualMemorySize;
  public ProcResourceValues(long cumulativeCpuTime, long physicalMemorySize,
                            long virtualMemorySize) {
    this.cumulativeCpuTime = cumulativeCpuTime;
    this.physicalMemorySize = physicalMemorySize;
    this.virtualMemorySize = virtualMemorySize;
  }
  /**
   * Obtain the physical memory size used by current process tree.
   * @return physical memory size in bytes.
```

```
   */
  public long getPhysicalMemorySize() {
    return physicalMemorySize;
  }

  /**
   * Obtain the virtual memory size used by a current process tree.
   * @return virtual memory size in bytes.
   */
  public long getVirtualMemorySize() {
    return virtualMemorySize;
  }

  /**
   * Obtain the cumulative CPU time used by a current process tree.
   * @return cumulative CPU time in milliseconds
   */
  public long getCumulativeCpuTime() {
    return cumulativeCpuTime;
  }
}

/**
 * Get the ResourceCalculatorPlugin from the class name and configure it. If
 * class name is null, this method will try and return a memory calculator
 * plugin available for this system.
 *
 * @param clazz class-name
 * @param conf configure the plugin with this.
 * @return ResourceCalculatorPlugin
 */
public static ResourceCalculatorPlugin getResourceCalculatorPlugin(
    Class<? extends ResourceCalculatorPlugin> clazz, Configuration conf) {

  if (clazz != null) {
    return ReflectionUtils.newInstance(clazz, conf);
  }

  // No class given, try a os specific class
  try {
    String osName = System.getProperty("os.name");
    if (osName.startsWith("Linux")) {
      return new LinuxResourceCalculatorPlugin();
    }
  } catch (SecurityException se) {
    // Failed to get Operating System name.
```

```
      return null;
    }

    // Not supported on this system.
    return null;
  }
}
```