Clotho: An Elastic MapReduce Workload/Runtime Co-design

Weiming Shi and Bo Hong
School of Electrical and Computer Engineering
Georgia Institute of Technology
Outline

- Introduction
  - Some Jargons
  - Problem with Multi-tenancy (P1)
  - Problem with MapReduce (P2)
  - When MapReduce meets Multi-tenancy (P1 & P2)
- Existing Works and Our Perspective
- Design and Implementation of Clotho
- Evaluation
- Summary
Introduction - Some Jargons

- MapReduce
  - A data-parallel distributed programming model and its associated execution runtime

- Hadoop
  - An open-source implementation of the MapReduce execution runtime

- Resource Container
  - A logical representation of the allocated computing resource (CPU, memory, and etc.)

- Resource Fragment
  - A logical representation of the allocated yet un-utilized computing resource
Introduction - Problem with Multi-tenancy (P1)

A multi-tenant cluster is prone to have resource fragments

- Root cause: unpredictable user demand
  
  Most users are ordinary users who have almost no idea of the resource demand of their programs
  
  If user programs are treated as “black boxes”, it is hard to infer their actual resource demand

- Dilemma: high resource utilization vs. high performance isolation
  
  Provisioning resource containers based on the worst-case estimation would create underutilized resource
  
  Conducting simple admission control based on the remaining resource and allowing the users to contend for the resource would cause the interference among them – misbehaving users
Introduction - Problem with MapReduce (P2)

A MapReduce execution pipeline is prone to have bottleneck stages.

- MapReduce execution runtime instantiates a distributed execution pipeline.
- Each execution pipeline can be decomposed into several consecutive stages:
  - e.g., Disk I/O -> CPU -> Disk I/O -> Network I/O -> Disk I/O -> CPU -> Disk I/O -> Network I/O
- Bottleneck occurs if the amount of assigned/induced work is allocated with an incommensurate amount of resource due to unpredictable user demand.
- A bottleneck stage could impede the upstream stages and starve the downstream stages.
Introduction – When MapReduce meets Multi-tenancy

- A MapReduce multi-tenant Cluster has both problems (P1 & P2)
- Special characteristics of MapReduce workload may give us some opportunities
  - It is executed by a pipelined execution runtime
  - Each pipeline stage has a somewhat ‘predictable’ resource utilization pattern
Outline

- Introduction
- Existing Works and Our Perspective
  - A Bimodal Pattern
  - Reflection on Existing Schemes
  - Our Perspective
  - Our Attempt
- Design and Implementation of Clotho
- Evaluation
- Summary
Existing Schemes – A Bimodal Pattern

- System-friendly scheme
  - Users provide a **static worst-case estimation** of the demand before the execution
  - System conducts the resource allocation based on the system capacity and this static estimation
  - Examples: slot-based Hadoop (a.k.a. MRv1), resource-based Yarn (a.k.a. MRv2)

- User-friendly scheme
  - Users provide almost no clue of the demand and hence user programs are “Black boxes”
  - System conducts the **resource monitoring** and calibrates the resource allocation decision
  - Examples: Some improved Hadoop extensions
Reflection on Existing Schemes

- Previous mindset has hinged on the availability of the static user demand or dynamic system monitoring and leaves the duty to either users or system.
- It mostly focused on provisioning the resource dynamically and adaptively.
- It does not consider the user demand or the resource utilization of the workload as a tangible **control knob**.
Our Perspective

- Efficient resource management of a multitenant MapReduce cluster should be a chorus of two parties but not a solo of either party
- Joint efforts from both users and system are the key to the success
- A better resource management decision can be achieved if we have
  - **Elastic container**: the system can dynamically shape the allocated resource container with better clues of the real resource utilization of the user programs
  - **Elastic workload**: the user programs can adapt the resource utilization based on the allocated resource
- The amount of work is conserved whereas the process rate could be changed
Our Attempt on the Elastic Workload Approach

An Elastic Workload/Runtime Co-design

- A resource management scheme that tries to address P1 & P2
- Currently it can only handle CPU-intensive MapReduce workload
Outline

- Introduction
- Existing Works and Our Perspective
- Design and Implementation of Clotho
  - Design Choice for P1
  - Design Choice for P2
  - Sketch of Implementation
- Evaluation
- Summary
Design of Clotho

P1: How do we ensure the allocated resource based on the static worst-case demand is fully utilized?

We choose to honor the performance isolation first for the benefit of users.

We advocate for elastic (malleable) MapReduce workload

- User input should be splittable (ideally divisible workload)
- We need a flexible (as opposed to fixed) number of worker threads per stage of the MapReduce execution pipeline so we can adapt the number of worker threads (resource demand) based on the allocated resource.
- We unleash the resource demand of MapReduce workload on a particular resource only when it is possible and beneficial for both the users to speed up the execution and for the system to alleviate the resource fragments.
Design of Clotho

P2: How do we detect the bottleneck stage in the MapReduce execution pipeline?

Not all the MapReduce workloads can expand/shrink the demand on a particular resource by changing the number of worker threads per stage of the execution pipeline.

Currently we only conduct the bottleneck detection on CPU resource:

- Profile several iterations of the user programs in sampled tasks and measure the time spent on each resource (I/O, CPU).
- Those stages whose execution time is much longer than others are suspected to be the bottleneck stages.
Sketch of the Implementation of Clotho

- Clotho builds upon the Hadoop (MRv1)
- A new Mapper Class: ElasticMapper
  - The number of threads per Mapper task (the CPU resource utilization) can be changed dynamically
- Local resource Manager (LRM) and Local resource agent (LRA)
  - LRA and LRM follows the same master-slave architecture of Child and TaskTracker of Hadoop
  - LRM leverages the performance counters of Hadoop runtime to monitor and collect the latest resource utilization information.
  - LRA conducts the bottleneck detection with a cost/benefit analysis and heartbeats with LRM to share the information of the user programs with LRM proactively
  - LRM coordinates with LRA to trigger ElasticMapper to spawn/retire worker threads to expand the CPU resource utilization
Cost/Benefit Analysis of Local Resource Agent

- The life cycle of a record at the Map stage
  - Record reading -> Mapper Execution -> Record Writing
- Collected aggregated time spent on each substage (ideally different resource)
  - Time on record reading: $M_r$
  - Time on mapper execution: $M_e$
  - Time on record writing: $M_w$
Cost/Benefit Analysis of Local Resource Agent

- The original total completion time $T_o$:
  \[ T_o = M_r + M_e + M_w \]

- The predicted total completion time after expanding the CPU resource utilization $T_p$:
  \[ T_p = \alpha M_r + \frac{1}{p} M_e + \alpha M_w + O_p \]
  - $p$: the number of worker threads per Mapper task
  - $\alpha$: the ratio of the amount of transformed input data to the amount of the original data
    - $\alpha = 1$ if the input data is splittable as no transform is conducted and hence no extra amount of data is read
    - $\alpha = p$ if the input data is bloated by a factor of $p$. This occurs when the original input data is not splittable but can be transformed into a splittable input data so that each extra worker thread has work to do..
  - $O_p$: the synchronization overhead with $p$ worker threads
Cost/Benefit Analysis of Local Resource Agent

- \[ T_p - T_o = (\alpha - 1)(M_r + M_w) + O_p - \left(1 - \frac{1}{p}\right)M_e \]

- To simplify the reasoning, we assume that the overhead \( O_p = \Theta(M_r + M_w) \)

- If \( M_e \gg M_r + M_w \), then clearly \( T_p < T_o \)
  - Spawning more threads per task would be beneficial
Outline

- Introduction
- Existing Works and Our Perspective
- Design and Implementation of Clotho
- Evaluation
  - Experiment Setup
  - Correctness of Cost/Benefit Analysis
  - Speedup and Utilization Improvement
- Summary
Evaluation - Experiment Setup

- Experimental System
  - A four-node cluster
  - Each node hosts a 12-core CPU, 48G memory, a 500GB SATA disk (3Gb/s 7200 RPM)

- System Configuration
  - One node is selected to be the centralized controller (*NameNode* and *JobTracker*)
  - Every node is a worker node (*DataNode*, *TaskTracker* and *LocalResourceManager*)
  - We configure a maximum 16 map slots and 8 reduce slots for our experiment system
  - This leaves out some spare resource to mimic the possible CPU resource fragments or light load scenario in the real production environment
Evaluation - Correctness of Cost/Benefit Analysis

- Benchmark: Padded WordCount
  - It is a synthesized microbenchmark used to evaluate the correctness of the cost/benefit analysis of LocalResourceManager.
  - We pad a varied number of CPU-intensive operations into the Map function of WordCount to create the varied CPU resource demand.
  - We run the experiments with the stock WordCount (our baseline) and our ElasticMapper – enabled WordCount.
Evaluation – Correctness of Cost/Benefit Analysis

- Spawning more threads doesn’t help that much if the CPU consuming stage is not the bottleneck
- However, it does help when the time spent on CPU is much longer than the time spent on I/O

- $B(x,y)$: the stock WordCount (baseline)
- $E(x,y)$: the ElasticMapper-enabled WordCount
  - $x$: # threads
  - $y$: # iterations of CPU-intensive padding
Evaluation – Correctness of Cost/Benefit Analysis

- Spawning more worker threads doesn’t need shorter execution time.
- It comes with the extra overheads.
- It’s better to make sure it can help speed up the execution time before triggering the expansion.
- The cost/benefit analysis of LocalResourceManager fulfills this goal and can effectively predict the usefulness of spawning more worker threads.

- Elastic(x): the ElasticMapper-enabled WordCount
- Base: the stock WordCount (baseline)
- M(y): the group with y iterations of CPU-intensive padding
  - x: # threads
  - y: # iterations of CPU-intensive padding
Evaluation – Speedup and Utilization Improvement

- Distributed-Pentomino
  - It conducts a combinatorial search and features a CPU-intensive Map function.
  - It is a real MapReduce Application used to evaluate the improvement of the system utilization (P1) and the speedup of the user programs (P2).
  - We run the experiments with the stock Distributed-Pentomino (our baseline) and the ElasticMapper –enabled one.
  - We have transformed the input data to meet the splittable input requirement of ElasticMapper.
    - The original input has only one number which specifies the number of simulations for Distributed-Pentomino.
    - The original input is decomposed by the users of ElasticMapper into several numbers whose summation is equal to the original number of simulations.
    - The amount of work is still conserved.
Evaluation – Speedup and Utilization Improvement

- **B(x)**: the baseline with x single-threaded map tasks (y=1)
- **E(x,y)**: the ElasticMapper-enabled one with x map tasks and each task with y worker threads

- If we keep x the constant and double y from 1 to 4
  - The CPU utilization of the system increases from 50% to 75% and 95%.
  - A 1.18x and 1.44 x speedup for the entire application is observed.

- If we reduce the y but keep x*y constant,
  - Some extra speedup is observed (bump from 1.18x to 1.30x and 1.44x to 1.56x)
  - For x*y = constant, the number of potential work threads would be the same if the system has the resource fragments to accommodate the extra (y-1) threads, e.g. B(2001), E(1001,2) and E(501,4)
  - Smaller x has less task scheduling overheads and hence extra speedup
Summary

- Clotho is our preliminary work towards an elastic user/system cooperative resource management framework.
- It targets for reducing the resource fragments in the MapReduce system and meanwhile alleviating the bottleneck in the MapReduce execution pipeline by utilizing these resource fragments opportunistically.
- Currently it only handles CPU resource and Map stage bottleneck.
- We plan to add I/O as the first-class resource into our framework and apply the methodology to other stages in the MapReduce execution pipeline.
Questions?

Thank you!