OS-Assisted Task Preemption for Hadoop

Mario Pastorelli, Matteo Dell’Amico, Pietro Michiardi
EURECOM, France

DCPerf 2014
Madrid, 30 June 2014
Outline

1. Why Task Preemption On Hadoop
2. Our Approach
3. Experiments
Outline

1. Why Task Preemption On Hadoop
2. Our Approach
3. Experiments
Hadoop MapReduce

Bring the **computation to the data** – split in blocks across a cluster

**Map**

- **One task per block**
  - Hadoop filesystem (HDFS): typically, 64–512 MB
- **Stores locally key-value pairs**
  - *e.g.*, for word count: [(red, 15), (green, 7), ...]
Hadoop MapReduce

Bring the **computation to the data** – split in blocks across a cluster

**Map**

- **One task per block**
  - Hadoop filesystem (HDFS): typically, 64–512 MB
- Stores locally **key-value pairs**
  - *e.g.*, for word count: `[(red, 15), (green, 7), ...]`

**Reduce**

- **# of tasks** set by the programmer
- Mapper output is **partitioned by key** and **pulled** from “mappers”
- The Reduce function operates on all values for a single key
  - *e.g.*, `(green, [7, 42, 13, ...])`
High-Priority Tasks

- MapReduce jobs are made of several **tasks**
  - we will focus on the **task granularity**

- Sometimes you have **high priority tasks**
  - humans waiting for the results
  - high-value computations

- Some tasks may take **very long**
  - errors in implementation
  - simply, a lot of computation

- Solution: **preempt** low-priority tasks and give the resources they are using to high-priority ones
Preemptive Scheduling

- Priority can be **decided by a scheduler**
  - **fairness**: guarantee that no user can “cheat the system”
    [Zaharia et al., EuroSys 2010]
  - **deadline scheduling**: ensure jobs are completed by a **due date**
    [Kc and Anyanwu, CloudCom 2010]
  - **optimize response time**: let small jobs pass in front
    [Wolf et al., Middleware 2010; Pastorelli et al., BIGDATA 2013]
In Hadoop, Now

- Currently, Hadoop can only preempt tasks by **killing** them
  - waste work
- ...or you just **wait** for them to finish
  - introduce **latencies**
- We want to **do better**!
Outline

1. Why Task Preemption On Hadoop
2. Our Approach
3. Experiments
Delegating To the OS

- Hadoop tasks are **standard POSIX processes**
  - they communicate through **POSIX signals**
- We use the same strategy: **SIGTSTP, SIGCONT**
- Our implementation mirrors the one for **killing tasks in Hadoop**
  - SIGTSTP takes the place of SIGTERM
- The state of the computation is **implicitly saved by the OS**
  - will be **paged to disk** if necessary
The OS and Paging

- Memory is occupied by **running processes** and **file system cache**
- When it is full, pages are **evicted** from memory
  - Least Recently Used-like policy
  - Prioritizing **clean** pages (not modified after reading)
    - **don’t need page out**
  - Page out operations are clustered to **improve throughput**
    - disk seeks are amortized

**Thrashing**: when the **working set** (memory used by running programs) is larger than memory
In Hadoop, a best practice is to configure the OS to prioritize **running processes** over disk cache.

- Hadoop **reads in streams**, so cache is not important
- This **minimizes paging out**

Paging out is done **efficiently**

- close to maximum disk speed

**No Trashing!**

- suspended tasks are **not in the working set**
Outline

1. Why Task Preemption On Hadoop
2. Our Approach
3. Experiments
Experiments

Experimental Settings

- $t_l$, $t_h$: tasks with low and high priority
- Synthetic tasks parsing randomly generated data
- 512MB blocks
- We vary the arrival time of $t_h$
Standard Case

sojourn time $t_h$ (s) vs $t_l$ progress at launch of $t_h$ (%)

makespan (s) vs $t_l$ progress at launch of $t_h$ (%)
**Worst Case**

- Each job allocates 2GB of memory
  - it’s **a lot**, requires modifying the Hadoop configuration
## Overheads Due To Memory Usage

<table>
<thead>
<tr>
<th>Memory Allocated by $t_h$ (GB)</th>
<th>Swap Overhead (s)</th>
<th>Makespan Overhead (s)</th>
<th>Sojourn Time Overhead (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.625</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.875</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The diagram shows the overheads due to memory usage with respect to memory allocated by $t_h$, where:

- Green line: Swap overhead
- Blue dashed line: Makespan overhead
- Red dotted line: $t_h$ sojourn time overhead

The x-axis represents the memory allocated by $t_h$ in gigabytes (GB), while the y-axis represents the overhead in seconds. The graph illustrates how these overheads increase as more memory is allocated by $t_h$. The overheads are measured in megabytes (MB) for memory usage and seconds for time overheads.
Another Approach: Natjam

- **Natjam** [Cho et al., SoCC 2013] works at the **application layer**
- Requires **explicit handling** by the application:
  - currently works for **stateless** MapReduce programs
  - proposes hooks for serialization/deserialization to deal with state
Another Approach: Natjam

- **Natjam** [Cho et al., SoCC 2013] works at the application layer
- Requires **explicit handling** by the application:
  - currently works for **stateless** MapReduce programs
  - proposes hooks for serialization/deserialization to deal with state

### Pro
- Might **compress** the amount of data written to disk

### Con
- Would **always**, pessimistically, write to disk
- Requires serialization/deserialization overhead

- The two approaches can be **both available** to a scheduler
Resume Locality

- You can resume only **locally suspended tasks**
  - process migration could be implemented, but it would be **expensive**
  - ...or you could just restart the task from scratch

- **Delay scheduling** [Zaharia *et al.*, EuroSys 2010]: wait until a threshold before scheduling non-local work
  - can be done also here
Implications On Scheduling

- To optimize **wall time**, suspend tasks that are **closest to completion**
  - avoid **stragglers** (late tasks) as much as possible

- To **avoid redundant work**, suspend tasks with **smaller memory footprint**
  - avoid swapping overheads
Implementing Suspension-Friendly Tasks

Controlling Memory Footprint
- It could be worth to optimize for using less memory
- Hint the garbage collector to run on suspension
- Use garbage collectors that do deallocate RAM

External State
- Some tasks interact with the outside world
- Suspension should be handled correctly, but probably needs testing
Task preemption is **important for Hadoop scheduling**
- priorities, fairness, deadlines, size-based schedulers, ...

**We do not need to reinvent the wheel**
- OSes have been suspending processes for many years
- they **do it well**, let’s just **use them!**

**Swapping is not bad *per se***
- Hadoop mechanisms keep the working set under control and avoid thrashing