FMEM: A Fine-grained Memory Estimator for MapReduce Jobs

Lijie Xu†‡, Jie Liu†, and Jun Wei†

†Institute of Software, Chinese Academy of Sciences
‡University of Chinese Academy of Sciences
{xulijie09, ljie, wj}@otcaix.iscas.ac.cn

Abstract
MapReduce is designed as a simple and scalable framework for big data processing. Due to the lack of resource usage models, its implementation Hadoop hands over resource planning and optimizing works to users. But users also find difficulty in specifying right resource-related, especially memory-related, configurations without good knowledge of job’s memory usage. Modeling memory usage is challenging because there are many influencing factors such as framework’s dataflow, user-defined programs, large space of configurations and memory management mechanism of JVM. In order to help both users and the framework to analyze, predict and optimize memory usage, we propose a Fine-grained Memory Estimator for MapReduce jobs called FMEM. FMEM contains a dataflow estimator which can predict the data volume flowing among map/reduce tasks. Based on dataflow and rules of memory utilization learnt from a lot of jobs, FMEM uses a rules-statistics method to estimate fine-grained memory usage in each generation of task’s JVM. Representative benchmarks show that FMEM can predict diverse jobs’ memory usage within 20% relative error. Furthermore, FMEM will be promoted to find optimum dataflow and memory related configurations.

1 Introduction
Google MapReduce [7] framework and its open-source implementation Hadoop have been widely adopted to process big data. This framework divides the costly data processing job into small independent map/reduce tasks and runs them in parallel. Users only need to specify map and reduce functions to develop data-intensive applications, regardless of distributed issues. Users can also write SQL-like scripts which can be transformed into MapReduce jobs automatically by high-level frameworks such as Pig [15], Hive [18] and Sawzall [16].

Although MapReduce helps users focus on job’s function implementation, we find its three-isolated-layer architecture causes users’ difficulty in configuration, resource planning and performance optimization. In user layer, users are required to write programs, prepare dataset and also specify appropriate memory-related configurations. In framework layer, besides defining job’s data processing steps (dataflow [5]) in map and reduce stage, framework is also responsible to schedule, launch and maintain map/reduce tasks. In execution layer, each task runs as a separate JVM instance, performs data processing steps and executes concrete map/reduce functions. Since JVM divides memory into small spaces and manages them separately, only execution layer knows the actual fine-grained memory usage. Framework just treats memory as a large contiguous space without modeling its consumption. So inappropriate configurations may lead to job’s OutOfMemory error, performance degradation or resource waste. At the highest layer and facing large space of configurations, users usually feel hard to analyze, predict and optimize memory usage. However, new scheduling frameworks such as YARN [6] and Mesos [12] not only require users to specify the memory usage but also schedule tasks according to it.

It is challenging to model and predict job’s memory usage with variable dataset, limited logs and large space of configurations. Fortunately, MapReduce dataflow pattern is relatively fixed with only black-box map/reduce functions. Our proposed memory estimator (FMEM) uses simulation method to model dataflow pattern and statistical methods to model intermediate data volume. Memory usage is more complex to model because of multiple factors such as dataflow, configurations and garbage collection (GC). In order to build this model, we integrate the different views of memory consumption in all layers, study the memory management mechanism of JVM, analyze a lot of jobs’ logs, and then summarize rules of fine-grained memory usage. Statistical methods are used to estimate the size of in-memory objects. Finally, FMEM profiles a job using sample data and then predict its dataflow and memory usage on real big data.

Our contributions are as follows: 1) We provide a detailed analysis of job’s memory usage, considering dataflow and memory management from user-level to inner JVM. 2) We also introduce a fine-grained memory estimator which can predict job’s memory usage in a large space of configurations.
2 Memory Usage Analysis

Each mapper/reducer runs as an independent process in MapReduce framework. In Hadoop, one process is one JVM instance which isolates the framework from managing the physical memory directly. Memory allocation and GC are controlled by specific algorithms. JVM divides the whole heap space into two parts: new generation for storing newly-generated objects and old generation for storing long-term objects. We find task’s memory consumption mainly comes from the following items:

Memory Buffers. In mapper, spill buffer always occupies a large fixed space in old generation. It is set by io.sort.mb and used to cache map() outputs. Enlarging this buffer may reduce spill times and disk I/O. In reducer, data shuffled from map outputs are kept as in-memory segments in a logical shuffle buffer. This buffer cannot exceed a threshold (default 70%) of JVM’s total heap, or else segments are merged onto disk. In JVM, segments are first allocated in new generation and some of them are transferred into old generation if GC occurs. In addition, Java’s input/output/flush/compress streaming classes contain small-sized buffers.

Records. Since each task has to read <K, V> records, process them, merge intermediate records and output new records, records definitely occupy a large space in JVM. In mapper, map() outputs records into spill buffer. In reducer, shuffled records are first kept as segments in shuffle buffer, though they may be merged onto disk later. Streaming records in map() and reduce() occupy limited space unless many of them are kept purposely into in-memory data structure.

Temporary Objects (TmpObjs). While processing and producing records, user-defined programs or framework itself may generate temporarily referenced objects such as char[], byte[], String, ArrayList and so on. Most of them are auxiliary objects of input/output records, allocated in new generation first and then reclaimed by GC. For example, A WordCount mapper produces massive java.nio.HeamCharBuffer objects. Objects’ number equals the number of map() output records, but their size is more than 7 times the size of map() input records.

Others. The native libraries used in task’s JVM may consume small memory space. JVM also keeps a small area to store programs’ Class, Object and Method information. Other program-related items such as code segment and thread pool also have small space in memory.

3 System Overview

To predict jobs’ Memory Usage on big job’s mu on big dataset (BData). Conf stands for Configuration.

Built-in Monitor: To monitor dataflow, we add many fine-grained dataflow counters into Hadoop’s task logs. For example, we add each spill piece’s Records/Bytes Statistics (RBS) before and after spilling, each partition’s RBS before and after merging and so on. We also use Jstat [4] to record each generation’s memory usage every N seconds. Users can turn on or off built-in monitor through configuration. This monitor has low overhead and only used for sample jobs.

Profiler: After a sample job finishes, log collector will fetch each task’s execution time, configuration, dataflow volume and memory usage. Dataflow profiler calculates task’s RBS in map, spill&merge, shuffle, sort and reduce phase. Similarly, memory profiler calculates max/min/average memory usage in each phase.

Dataflow Estimator: Though we can get RBS from the sample job, it is non-trivial to predict big job’s dataflow in a large configuration space. Many configurations such as input split size, spill buffer and reducer number can affect dataflow volume. To tackle them, we actually build a simulator of MapReduce framework to model dataflow in each processing step. Statistical methods are used to model and estimate the I/O ratio. When big job’s BData and Conf2 are specified, mapper dataflow model in our simulator uses sample mappers’ profiles to estimate new mappers’ dataflow. Then, reducer dataflow model can compute new reducers’ profiles based on the sample ones.

Memory Estimator: To estimate new tasks’ memory profiles, we first compute the size of their memory-consuming items. We get memory buffer size from Conf j2, get records’ size from dataflow estimator, and compute TmpObjs according to dataflow and memory profiles of sample tasks. Next, we use rules summarized from tremendous jobs’ profiles to estimate memory usage in each generation for each task. The rules are formalized as NGU/OGU = f (Conf, Records, TmpObjs). Finally, memory estimator selects the maximum (x)
memory usage of all the new mappers to represent mapper’s 
mu. In detail, mapper’s xNGU represents maximum 
memory usage in new generation of all mappers. 
So does reducer’s, xOU stands for that in old generation. 
XHeapU denotes heap usage (i.e., NGU + OU).

4 Evaluation

Because each task runs as an independent JVM instance 
and processes its own data, task’s memory usage is not so 
sensitive to cluster scale as job’s execution time. We 
evaluate FMEM’s accuracy on a local cluster of 10 n-
odes. Each node has four Intel i7-2600 cores, 16GB 
RAM and 2TB disk space. OS is Ubuntu-11.04 x86_64 
and JDK is HotSpot 64-Bit Server VM (build 1.6.0_27). 
Hadoop version is 0.20.2 which is similar to the latest 
1.2. YARN does not change dataflow pattern either. 
One node act as JobTracker. The others are slave nodes, 
each of which has 4 map slots and 2 reduce slots.

We use diverse applications (Table 1) to evaluate 
FMEM. Combine denotes whether combine() is used. Compress denotes whether spill pieces and segments are 
compressed. SeqBlock means Block compression in Se-
quenceFile. For each application, we run 180 sample 
jobs (processing 1GB sample dataset) and 180 big job-
s (processing big dataset) with different combinations 
of <split, ismb, RN, Xmx, Xms> (SIRXX). These five 
configurations are often adjusted to better performance, 
though our models involve many other configurations. 
S-plit is input split size (set to 64, 128 or 256MB). ismb is 
io.sort.mb (set to 200, 400, 600 or 800MB). Sample job-
s’ RN (reducer number) is 2 or 4, while big jobs’ RN is 
9 or 18. JVM’s maximum heap size Xmx is set to 1000, 
2000, 3000 or 4000MB. Minimum heap size Xms is not 
set or set equal to Xmx. So the number of sample/big 
jobs is 192. Twelve of them are abortive jobs because 
of memory overflow. Next, we use a sample job with 
specific <split, ismb, RN, Xmx, Xms> to estimate a big 
job’s mu with another SIRXX. So there are 180 * 180 = 32,400 estimated memory usage <emu>. Finally, we 
compare each big job’s estimated <emu> and real <emu> 
using relative error as follows:

\[ \text{relative error} = \frac{|\text{emu} - \text{mu}|}{\text{emu}} \times 100\% \]

If rmu = 0, we set relative error to 100%. The sample 
job randomly selects several splits (totally 1GB) from all 
the input splits of big dataset as sample dataset.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Dataset</th>
<th>Combine</th>
<th>Compress</th>
</tr>
</thead>
<tbody>
<tr>
<td>WikiWordCount</td>
<td>9.4 GB</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>BuildInvertedIndex</td>
<td>9.4 GB</td>
<td>N</td>
<td>SeqBlock</td>
</tr>
<tr>
<td>UserVisits_Aggre-pig</td>
<td>75 GB</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>TwitterBiEdgeCount</td>
<td>24.4 GB</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>TeraSort</td>
<td>36 GB</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

WikiWordCount (WWC): This application uses stan-
dard WordCount program from Hadoop Examples. We 
preprocess enwiki-20100405-pages-articles.xml and get 
9.4GB plain text as input big dataset.

BuildInvertedIndex (BII): This application simulates 
building inverted index of Web pages, which is widely 
used in search engines. The source code is from [1]. In-
put dataset is as same as that in WWC.

UserVisits_Aggre-pig (UVA): This application is actu-
ally a Pig script which is used to analyze user-visited logs 
in websites. We get this script from Hive Performance 
Benchmark in [3]. It has Group By operator and uses 
program-generated dataset.

TwitterBiEdgeCount (TBEC): It counts the number of 
bilateral edges of Twitter graph from [13]. This large 
sparse graph has more than 40 million nodes and 1.5 bil-
lion edges.

TeraSort (TS): This application also uses standard 
TeraSort program and sorts program-generated 36 GB 
dataset. Note that this job uses identity map() and re-
duce(). Thus, the I/O ratio of them is 1:1.

4.1 Evaluating Memory Estimator

Each job’s memory usage is represented by mapper’s 
mu and reducer’s mu. We evaluate them separately. 
each histogram in Figure 2 shows the average relative 
error from 32,400 comparisons of big jobs’ <emu, r-
umu>. Four metrics (xOU, xNGU, xHeapU and RSS) 
are used as concrete mu for both mappers and reduc-
ers. Suppose a big job has n mappers, this job’s mapper 
xOU = max_{1 \leq i \leq n} (OU_i). Others are computed 
in the same way. HeapU represents total memory usage 
of JVM, while RSS (Resident Set Size) stands for non-
swapped physical memory usage in Linux. Sometimes 
there is a small difference between them. The top part 
shows mapper’s relative error. Compared with xOU, xN-
GU has higher error rate. One reason is that NGU is more 
variable and affected by multiple factors. Another is that 
our estimating condition is very harsh. We only use a 
single sample job with one configuration to estimate a 
big job with another configuration. xHeapU and RSS are 
better but their standard deviations are a little high. The 
bottom part shows reducer’s relative error. Since reduc-
er’s mu is related to the size of shuffled records, large d-
ifference of dataflow may cause high error rate of mu. So 
WWC’s xOU and xNGU have high error rate. But for the 
other applications, xNGU and xOU have low error rates 
which indicate our memory usage rules are effective.

5 Related Work

Many researchers have studied job’s performance mod-
el and optimizing methods. Some are concerned about
FMEM which can estimate MapReduce job’s dataflow analyzing, predicting and optimizing resource usage. To help users analyze, predict and optimize resource usage, we develop FMEM which can estimate MapReduce job’s dataflow usage. To help users analyze, predict and optimize resource usage, we develop FMEM which can estimate MapReduce job’s dataflow carefully. They also discuss how to allocate right resource (slots) to guarantee job’s runtime. Other researchers optimize job’s configurations. Starfish project [11, 10] proposes a cost-based optimizer to find job’s optimum configuration. The What-if engine in this project can predict job’s performance (mainly for runtime) with different configurations. Hadoop performance models are discussed in [9] but fine-grained memory usage is not studied.

Few works focus on job’s memory usage. Singer et al. [17] design a fork-join MapReduce Java Framework (MRJ) for multi-core machines. They use machine learning approach to finding most suitable GC policy for MRJ, but memory usage is not studied. This method does not concentrate on distributed MapReduce framework like Hadoop either.

6 Conclusion

Memory is more precious compared with disk for big data processing. YARN and Mesos schedule tasks according to CPU and memory requirement. To help users analyze, predict and optimize resource usage, we develop FMEM which can estimate MapReduce job’s dataflow and memory usage in a large configuration space. It uses sample job’s profiles to estimate big job’s resource usage. FMEM models the complex relationship among dataflow, memory usage, GC and configurations. It can also be promoted to tackle other resource-related problems. To the best of our knowledge, this is the first approach that tries to model the memory usage of distributed MapReduce tasks. Our project is now available at github [2].

7 Acknowledgement

We are indebted to our reviewers and Xiulei Qin for their insightful comments. This work is supported by the National Natural Science Foundation of China (61202065, 61173005), the National Grand Fundamental Research 973 Program of China (2009CB320704), and the National High-Tech R&D Plan of China (2012AA011204).

References