LETTER

Eager Memory Management for In-Memory Data Analytics*

Hakbeom JANG[†a)], ***Student Member***, **Jonghyun BAE**[††], **Tae Jun HAM**[††], ***Nonmembers***, ***and*** **Jae W. LEE**[††], ***Member***

SUMMARY This paper introduces *e-spill*, an eager spill mechanism, which dynamically finds the optimal spill-threshold by monitoring the GC time at runtime and thereby prevent expensive GC overhead. Our *e-spill* adopts a slow-start model to gradually increase the spill-threshold until it reaches the optimal point without substantial GCs. We prototype *e-spill* as an extension to Spark and evaluate it using six workloads on three different parallel platforms. Our evaluations show that *e-spill* improves performance by up to 3.80× and saves the cost of cluster operation on Amazon EC2 cloud by up to 51% over the baseline system following Spark Tuning Guidelines.

key words: *in-memory computing, spark, garbage collection, data spill*

1. Introduction

Modern in-memory data analytic frameworks such as Apache Spark [1] and Ignite [2] are rapidly gaining popularity with their ability to provide orders of magnitude performance improvements over Hadoop MapReduce [3] on workloads with frequent data reuse (e.g., iterative algorithms). However, this performance gain is reduced when the memory footprint exceeds an available memory size. For example, in case of Spark, such scenario can lead to a significant amount of garbage collection (GC) operations which can account for nearly 50% [4] of an execution time, thus incurring more than a 2x system slowdown.

Previous proposals address this challenge by i) adjusting the working set size by tuning task granularity and parallelism [5] or ii) moving large objects to outside the heap (i.e., JVM heap) [6]. In addition, the conventional in-memory processing systems provide *spill-mechanism* that serializes the partially created data and writes it to the local disk. Each running task (i.e., thread) estimates the size of objects created at runtime and triggers a spill operation when the estimated volume of the task reaches a certain spill-threshold, thus avoiding expensive GC overhead (especially for major GC). This feature not only alleviates the memory pressure but also improves the performance of system by avoiding time-consuming GC operations.

However, this benefit is not always ensured. One critical issue is that the actual size of objects in Java Virtual Machine (JVM) does *not* match the *estimates* used by the upper-layer in-memory processing frameworks. To confirm this point, we run a standalone Spark and measure an execution time across varying spill-thresholds on a 4-node homogeneous cluster. Figure 1 shows an execution time breakdown of the reduce stage in Intel HiBench TeraSort workload. By default, in Spark 2.1.0, the spill-threshold is set to 60% of the JVM heap size (equal to Spark memory). However, as shown in the figure, the run with the default spill-threshold (the leftmost bar) still incurs considerable GC time. Towards the right, as the spill-threshold decreases, the GC time quickly reduces due to the reduced memory pressure. On the other hand, a frequent spill also means that each task needs to unnecessarily spill more data, resulting in more compute times. In this case, default spill-threshold ×1/16 reaches its optimal point in terms of the total task execution time.

This work introduces *e-spill*, an eager spill mechanism, which dynamically finds the optimal spill-threshold by monitoring the GC time at runtime and thereby preventing expensive GC operations. The proposed *e-spill* adopts a slow-start model to gradually increase the spill-threshold until it reaches the optimal point without substantial GCs. We prototype *e-spill* as an extension to Spark and evaluate it using six workloads on three different platforms: (1) a 4-node homogeneous cluster with 64 fat cores (Intel Xeon), (2) a single-node Intel Knights Landing (KNL) machine with 64 thin cores (Intel Xeon Phi), and (3) a virtualized 64-node Spark cluster on Amazon EC2 with 256 fat cores (Intel Xeon). The proposed *e-spill* achieves a geomean speedup

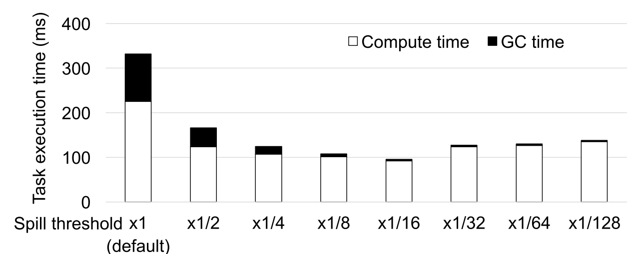

Fig. 1 GC overhead when varying spill threshold

Manuscript received September 13, 2018.
Manuscript revised November 7, 2018.
Manuscript publicized December 11, 2018.
[†]The author is with the College of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea.
[††]The authors are with the Dept. of Computer Science and Engineering, Seoul National University, Seoul, Korea.
*This work was supported by a research grant from Samsung Electronics, by IDEC (EDA tool), and by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. B0101-17-0644, Research on High Performance and Scalable Manycore OS).
a) E-mail: hakbeom@skku.edu
DOI: 10.1587/transinf.2018EDL8199

Copyright © 2019 The Institute of Electronics, Information and Communication Engineers

of 1.71× on a 4-node homogeneous cluster and 1.36× on a single-node KNL machine. Furthermore, *e-spill* achieve a geomean speedup of 1.30× and reduces the operating cost by 23% on a virtualized 64-node cluster.

Our contributions can be summarized as follows:

- Analysis of the spill mechanism of Apache Spark [1], a popular in-memory data analytic framework, as a knob to control GC overhead
- Design and implementation of *e-spill* on Spark, which dynamically finds the optimal spill-threshold by monitoring the GC time at runtime and avoids excessive GC operations
- Detailed evaluation and analysis of *e-spill* performance on three different parallel platforms

2. *e-spill*: Eager Spill Mechanism

The proposed *e-spill* is a low-cost runtime framework that finds the optimal spill-threshold for in-memory processing frameworks, to provide robust performance for various platforms without requiring workload-dependent information.

2.1 Background and Overview

Data spill is the process of storing partially created intermediate result to a local disk during task execution to prevent expensive GCs and *OutOfMemoryError*s resulting from the lack of Spark execution memory. A user program in Spark is described as a sequence of operations on Resilient Distributed Datasets (RDDs), which are the primary data abstraction for Spark. A spill operation can occur if all data has to be collected in a single buffer to create a shuffle file, or if multiple RDD partitions need to temporally store intermediate results to generate one RDD (e.g., `Join` and `Zip`). Initially, Spark allocates a small buffer (e.g., 5MB) to store the intermediate result. When the size of the buffer becomes insufficient, it doubles the size of the buffer. The maximum size that the buffer can reach is the total Spark execution memory divided by the number of currently running Spark worker cores. When a spill operation occurs, it serializes key-value pairs stored in the buffer one by one and flushes them to a spill file in a certain batch unit (default: 10000 key-value pairs). After all key-values pairs are written, Spark manages the spill file as a list and allocates a new buffer for the remaining key-value operations. This process is repeated until all the key-values of the partition are computed. After all the key-value operations are complete, the intermediate result stored in the spill file is merged with the remaining results in memory. The final result is stored in the shuffle file or is passed to the next operation.

Ideally, it is possible to avoid time-consuming major GCs by spilling objects which are located in the old-generation heap space of JVM before the heap is full. However, in reality, the estimator often mis-predicts the volume of the task (i.e., key-value buffer) and thus cannot effectively avoid such major GCs. From JVM's perspective, the spilled

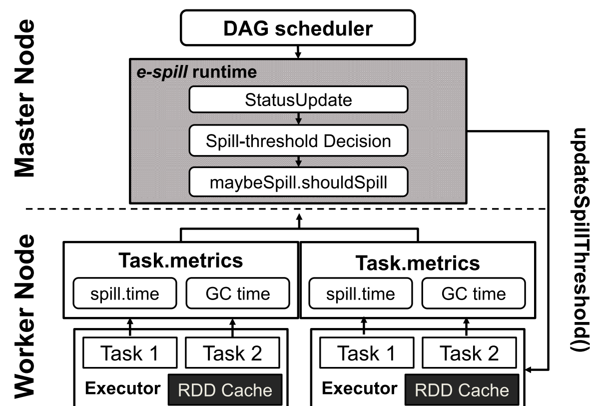

Fig. 2 *e-spill* overview

Algorithm 1 *e-spill* runtime

INPUT: JVM heap size, # of available CPU cores, spill time, GC time
OUTPUT: Spill-threshold

```
1:  while Key-value iterator has next item do
2:      if First key-value pair is inserted then
3:          Estimate size of first key-value pair
4:          Set initial spillThreshold ∝
5:            JVM heap size / # of Spark threads / key-value size
6:      end if
7:      Insert key-value pair in temporal buffer
8:      if # of key-value pair > spillThreshold then
9:          Spill occurred
10:         if gcTime > spillTime * 0.1 then
11:             Update spillThreshold / 2
12:         else
13:             Update spillThreshold * 1.25
14:         end if
15:     end if
16: end while
```

object is located in the old-generation heap space of JVM and this object is not freed until the entire spill process is completed. Meanwhile, Spark continues to generate small heap objects, which occasionally get promoted to the old-generation heap, and requests extra space there. This triggers frequent major GCs [5].

The proposed *e-spill* mainly aims to avoid frequent major GCs caused by spill operation due to the mis-prediction. Figure 2 shows the overall structure of *e-spill*, which extends the existing Spark's spill-mechanism shaded in gray. The *e-spill* runtime collects the *spill time* and *GC time* during task execution to monitor the time spent on GC caused by spill operations. To realize this, *e-spill* maintains a feedback loop between the master node and worker nodes. Finally, *e-spill* determines whether to increase or decrease a spill-threshold based on the ratio of *spill time* to *GC time*.

2.2 Runtime Algorithm

The proposed *e-spill* runtime starts with a calibration phase in which a slow-start model gradually increases the spill-threshold to find the optimal threshold without GCs. Algorithm 1 shows the *e-spill* runtime algorithm. After estimating the first <K,V> pair size, this algorithm determines an

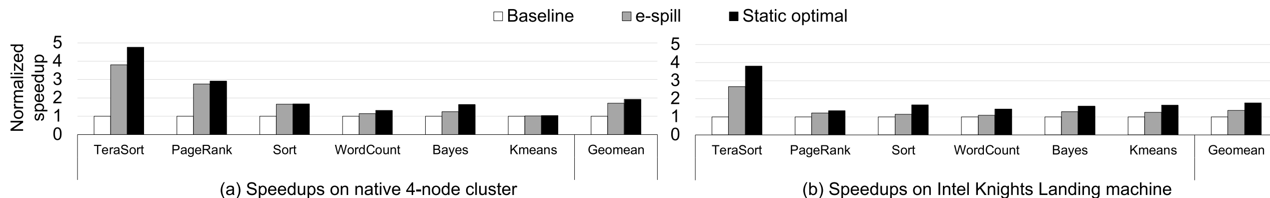

Fig. 3 Normalized speedups on two platforms: (a) native 4-node cluster (b) manycore platform

Table 1 Setup for three evaluation platforms

	Native cluster	Knights Landing (KNL)	Amazon EC2 cluster
CPU	Intel Xeon E5-2640v3 × 2 sockets	Intel Xeon Phi 7210	Intel Xeon E5-2676v4
Memory	16GB DDR4 × 8	16GB MCDRAM & 32GB DDR4 × 6	16GB
Disk	NVMe SSD 1.6TB	NVMe SSD 1.6TB	EBS 100GB
Network	40Gbps InfiniBand	1Gbps Ethernet	10Gbps Ethernet

Table 2 Workload characteristics

Workloads	Data Size		
	Cluster	KNL	AWS
TeraSort	128GB	80GB	1TB
PageRank	pages: 25M	pages: 15M	pages: 100M
Sort	128GB	80GB	1TB
WordCount	240GB	128GB	5TB
Bayes classification	pages: 20M classes: 20K	pages: 10M classes: 10K	pages: 160M classes: 160K
Kmeans	samples: 200M	samples: 150M	samples: 400M

initial spill-threshold based on a given JVM heap size and the number of Spark worker cores. As the task executes, the *e-spill* gradually increases the spill-threshold until excessive GCs does not occur. If a spill occurs, the threshold is halved if the ratio of the time spent on spill to GC is greater than 10%. Otherwise, if the time spent on GC is short enough (less than 10% of the time spent on spill), *e-spill* increases the spill-threshold to 1.25× the current threshold to optimize memory usage. After a round of operation is complete, the optimal spill-threshold found in the previous round is utilized in the next round to avoid redundancy. Since the characteristics of functions (transformations) are different across computation phases (i.e., stages), *e-spill* does not reuse the spill threshold across different stages and searches a new spill-threshold from scratch.

3. Methodology

We run workloads from Intel HiBench 5.0 [7] on three different parallel processing platforms as shown in Table 1. Table 2 summarizes inputs for the six applications. We compare *e-spill* to two designs:

- **Baseline:** This is a vanilla Spark following Spark Tuning Guidelines [8]. Each worker node has four Spark executors, and each executor runs 4 threads with a 5GB memory. In the KNL platform, we use four executors and each executor runs 16 threads with a 20GB memory. Lastly, in the virtualized cluster platform, each executor runs four threads with a 10GB memory.
- **Static Optimal:** We perform an exhaustive search to find the best-performing partition count for each stage. Such optimized partition count does not incur neither spills nor GCs. This number serves as the (impractical) theoretical maximum performance.

4. Evaluation

4.1 Program Speedups

On 4-node Homogeneous Cluster. Figure 3 (a) compares the speedups of two designs over the baseline configuration. The proposed *e-spill* achieves a geomean speedup of 1.71×, with a maximum speedup of 3.80×. More importantly, *e-spill* achieves the robust performance comparable to the static optimal configuration.

The first three applications in Fig. 3 (a) are shuffle-heavy workloads, which frequently trigger major GCs. TeraSort and Sort use `sortByKey` transformation, which results in the shuffle of the large data (i.e, entire RDD) between tasks. PageRank is an iterative algorithm that joins the intermediate output and a cached RDD and thus have a large memory footprint. In general, shuffle-heavy workloads utilize a large amount of memory. The primary source of performance improvement in *e-spill* is the reduced overhead of expensive GC operations. Figure 4 shows a task execution time breakdown of the map and reduce stage for TeraSort (shuffle-heavy). The proposed *e-spill* reduces the time spent on GC in TeraSort by 91% with only a modest increase in spill time.

The remaining three applications are classified as shuffle-light workloads. Their memory footprints are relatively small since these applications trigger the shuffle during a summary transformation, such as `reduceByKey`. For WordCount and Bayes, *e-spill* achieves speedup of 1.15× and 1.25×, respectively. In case of Kmeans, the baseline performs well on the cluster and thus all designs show similar performance. Overall, *e-spill* obtains the performance close to the performance of the static optimal configuration.

On Single-Node Manycore Machine. Figure 3 (b) shows the speedups of *e-spill* on a 64-core Intel KNL platform. For this platform, we use smaller inputs to reduce task failures, which occur frequently with the original input size as shown in Table 2. Furthermore, since several programs do not run to completion with the baseline configuration [8], we use a slightly-tuned baseline configuration (i.e., increased number of partitions) which allows the workloads to complete without experiencing significant major GC overhead. Since the

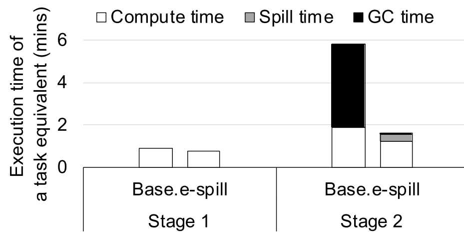

Fig. 4 Execution time breakdown for TeraSort

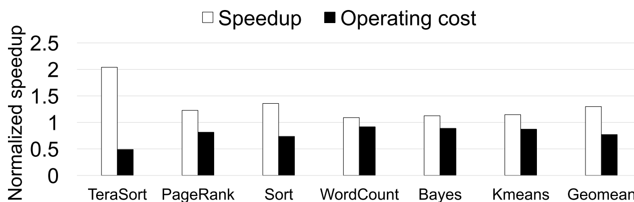

Fig. 5 Normalized speedups on cloud for 64-node Amazon EC2 cluster

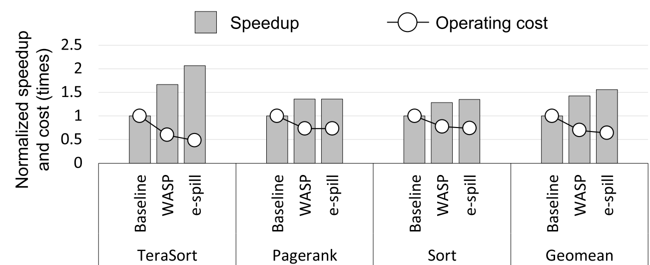

Fig. 6 WASP scheduler [5] vs. *e-spill*

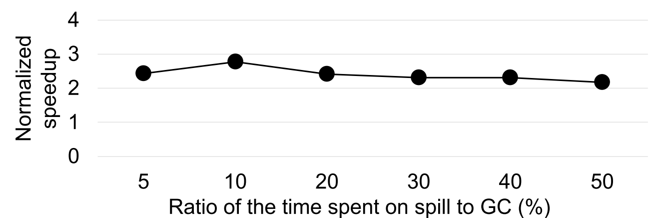

Fig. 7 Speedups across varying ratio of the time spent on spill to GC. All numbers are normalized to the baseline.

input size and the baseline configuration are different from those of the 4-node homogeneous cluster, the static optimal trend is different in this configuration. Overall, *e-spill* also works well on the single-node manycore system with 64 thin cores for all workloads with a geomean speedup of 1.36× and a maximum speedup of 2.68×.

On 64-node Amazon EC2 Cluster. We evaluate *e-spill* on a 64-node Amazon EC2 cluster with 256 fat cores to confirm the robustness of it on a large scale cluster. Figure 5 shows performance improvements and cost reductions from *e-spill* on a 64-node cluster. The result shows that *e-spill* achieves a geomean speedup of 1.30× and reduces the operating cost by 23%. Note that the operation cost is the financial cost of running Amazon EC2 cluster for the required execution time. To calculate the operating cost, we calculate the cost of running the 64 m4.xlarge instances. Each node consumes $0.246/h and $0.016/h for storage. We multiplied these values by the execution time to compute the cost.

4.2 Performance Analysis

Comparison with WASP scheduler. We compare *e-spill* with WASP, a state-of-the-art Spark task scheduler [5]. WASP jointly optimizes both task granularity and parallelism based on workload characteristics. WASP requires calculation of *memory amplification factor* (MAF) of each transformation function from various workloads to predict the memory usage of each stage with the goal of avoiding GC overheads. Figure 6 shows the speedup of WASP and *e-spill* on a 64-node virtualized Spark cluster with three shuffle-heavy workloads. Unlike WASP, *e-spill* does not require any offline profiling and outperforms it by up to 1.24×. The main source of improvement of *e-spill* is the reduced shuffle overhead. For example, we observed that WASP increases the number of partitions to 512-16384 between map and reduce stage in TeraSort. Such increase (i.e., a single partition gets smaller), leads to a reduction in the memory usage, and thus can eliminate GC overheads. However, such increase in the number of partitions also incurs substantial overhead for shuffle operations [9]. Since *e-spill* does not increase the number of partitions, *e-spill* achieves more robust performance than WASP for shuffle-heavy workloads.

Sensitivity on spill/GC ratio. Figure 7 shows average speedups of three shuffle-heavy workloads across varying ratio of the time spent on the spill to GC. Our experiment demonstrates that 10% is the optimal value.

5. Conclusion

This paper introduces *e-spill*, an eager spill mechanism, which dynamically finds the optimal spill-threshold by monitoring a GC time during a runtime. Our *e-spill* achieves a robust performance on three different parallel platforms without requiring any workload-dependent tuning parameters. Our evaluation on Spark shows that *e-spill* achieves a geomean speedup of 1.71× on a 4-node homogeneous cluster and 1.36× on a single-node KNL machine. Furthermore, *e-spill* achieve a geomean speedup of 1.30× and can reduce the operating cost of a virtualized 64-node cluster by 23%.

References

[1] "Apache Spark." http://spark.apache.org/.
[2] "Apache Ignite." https://ignite.apache.org/.
[3] "Apache Hadoop." http://hadoop.apache.org/.
[4] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu, "Yak: A high-performance big-data-friendly garbage collector," Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, pp.349–365, 2016.
[5] J. Bae, H. Jang, W. Jin, J. Heo, J. Jang, J.-Y. Hwang, S. Cho, and J.W. Lee, "Jointly optimizing task granularity and concurrency for in-memory mapreduce frameworks," 2017 IEEE International Conference on Big Data (Big Data), pp.130–140, Dec. 2017.
[6] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, "Facade: A compiler and runtime for (almost) object-bounded big data applications," Proceedings of the Twentieth International Conference on Ar-

chitectural Support for Programming Languages and Operating Systems, ASPLOS ’15, New York, NY, USA, pp.675–690, ACM, 2015.
[7] “Intel HiBench.” https://github.com/intel-hadoop/HiBench.
[8] “Apache Spark: Tuning Spark.” http://spark.apache.org/ docs/latest/ tuning.html/.
[9] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M.J. Freedman, “Riffle: Optimized shuffle service for large-scale data analytics,” Proceedings of the Thirteenth EuroSys Conference, EuroSys ’18, New York, NY, USA, pp.43:1–43:15, ACM, 2018.