Profiling Memory Vulnerability of Big-data Applications

N. Rameshan*†, R. Birke‡, L. Navarro*, V. Vlassov†, B. Urgaonkar§, G. Kesidis§, M. Schmatz‡, and L. Y. Chen‡
*Universitat Politècnica de Catalunya, Barcelona, Spain, Email: {rameshan,leandro}@ac.upc.edu
†KTH Royal Institute of Technology, Stockholm, Sweden, Email: {rameshan,vladv}@kth.se
‡IBM Research Zurich Lab, Rüschlikon, Switzerland, Email: {bir,mrt,yic}@zurich.ibm.com
§Pennsylvania State University, PA, USA, Email: {bhuvan,gik2}@cse.psu.edu

***Abstract*—Motivated by the increasing popularity of hosting in-memory big-data analytics in cloud, we present a profiling methodology that can understand how different memory subsystems, i.e., cache and memory bandwidth, are susceptible to the impact of interference from co-located applications. We first describe the design of the proposed tool and demonstrate a case study consisting of five Spark applications on real-life data set.**

I. Introduction

In-memory big-data analytics, such as Spark and Flink, have been widely adopted to process large amounts of data and to derive operational insights for the society and business. While they are typically hosted in a private environment, i.e., not co-located with other applications, the recent trend is to migrate big-data applications on cloud, leveraging the advantage of resource elasticity. The challenges raised immediately are how to ensure the performance dependability for big-data analytics in the cloud, especially given the existence of the notorious stragglers whose execution times are significantly longer than the other tasks. Due to sharing the underlying memory subsystem, e.g., cache and memory bandwidth, with co-located applications, the performance of big-data analytics can be highly volatile and mitigating their performance anomalies in cloud becomes further exacerbated. Profiling is the first step towards understanding performance variations, which in turn aids in enhancing the dependability.

It is long deemed challenging to understand the applications' sensitiveness particularly to memory interferences, due to the intricate dependency on cache, memory capacity and bandwidth. Profiling the applications [1], [2], [3] in an isolated and emulated environment, i.e., creating interference on certain components, can gain a good understanding of applications' characteristics of different memory subsystem and their adaptability to different levels of interference. Due to the distributed nature of big-data analytics, the degree of performance degradation highly depends on the intensity as well as the distribution of interference. It is important to consider the data and task distribution together with different interference patterns when analyzing the memory vulnerability of big-data applications.

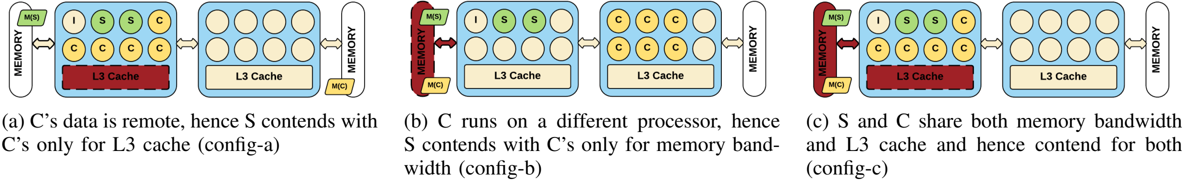
It is known that modern big-data analytics not only have a large number of tuning parameters but also a daunting number of performance metrics, due to its distributed nature. To correlate the interference with job performance, one needs to analyze the numerous metrics collected from the application as well as the underlying system. Take an example of analyzing the execution of k-means on a Spark cluster. The application latency is determined by the execution of numerous tasks over multiple stages, which can be further decomposed into scheduling, fetching, garbage collection and execution time. From each of the servers, one needs to collect application metrics and low-level metrics related to memory subsystem, e.g., last level cache misses, bandwidth usages, prefetches, etc to gain any understanding of interference. Though big-data analytics provides an interface to collect application metrics, there is a lack of tools that can aggregate metrics collected from distributed tasks and servers and provide insightful analysis on the memory sensitivity.

In this paper, we aim to provide an integrated solution to understand how robust or susceptible those business critical big-data applications are to potential memory interference in the cloud. We envision our analysis to serve as an oracle to guide the configuration of big data applications and the choices of hosting platforms, prior migrating to the wild cloud. To this end, we develop a tool chain that can: (i) emulate interference on different parts of the memory sub-system, i.e., memory bandwidth, cache, and both, including a set of tunable parameters on interference intensity, type and distribution along with configuration tuning of Spark; (ii) integrate metrics collection from both the distributed application and server; and (iii) correlate interference and latency degradation via a top-down analysis.

We present preliminary and small-scale results on applying such a tool chain on analyzing five Spark applications, namely, word count, k-means, naive Bayes, decision tree and triangle count. Our testbed is a small cluster consisting of three physical servers. We collect the performance degradation of these applications, in combination with different kinds, intensities, and distributions of interference. Our preliminary findings based on these five applications are: (i) the application performance indeed is degraded, but not to a great extent, i.e., roughly 25-35%, except triangle count which suffers a degradation of about 150%; (ii) these Spark

978-1-5090-3688-2/16 $31.00 © 2016 IEEE
DOI 10.1109/DSN-W.2016.58

IEEE computer society



(a) C's data is remote, hence S contends with C's only for L3 cache (config-a)

(b) C runs on a different processor, hence S contends with C's only for memory bandwidth (config-b)

(c) S and C share both memory bandwidth and L3 cache and hence contend for both (config-c)

Fig. 1: Configurations for generating contention at different resources. S denotes the Spark application and M(S) denotes the memory allocation of S. C denotes the co-runners and M(C) denotes the memory allocation of C. I denotes the core serving interrupts.

applications are more sensitive to cache interference than memory bandwidth; (iii) the Spark scheduler is adaptive to interference, to a certain extent; (iv) performance interference exaggerates the role of garbage collection and shuffling causing significant performance degradations; (v) smaller data partitions enhance the resistance against memory-related interference; and (vi) data locality does not play a major role in the presence of performance interference.

In the following, we give an overview on memory interference profiling in Section II, top-down sensitivity analysis in Section III, and a case study on the five Spark applications in Section IV.

II. Profiling Memory Subsystem Interferences

In this section, we detail how to inject a particular pattern of memory sub-system interference by using co-located applications, termed *co-runners*, and techniques to collect statistics of interests. As our focus here is to understand how big-data application react to different co-located applications, we need to collect performance counters of the Spark application as well as the co-runners.

We ensure CPU isolation by *co-running* Spark with other applications on different cores of the same physical host. We denote the Spark application by *S* and each co-runner by *C*. In our experiments, we compute the performance degradation D as the relative increase in total execution time of Spark running in isolation T_i and with co-runners T_c, i.e.: $D = \frac{T_c - T_i}{T_i}$.

A. Emulating interferences on memory subsystems

Performance interference at the memory subsystem manifests when there is contention for any of the shared resources such as cache or memory bandwidth. An application sensitivity to contention is primarily determined by how much an application progress benefits from its reliance on a particular shared resource. For example, an application heavily benefiting from cache will suffer a high degradation when there is contention for cache from co-runners, and the application is said to be highly sensitive to cache interference.

In order to perform an accurate analysis of Sparks' sensitivity to different shared memory resources, it becomes imperative to stress the shared resources individually with different intensities. It is challenging to design micro-benchmarks that stress the shared memory resources individually. For example, designing a micro-benchmark to stress the memory bandwidth in isolation requires careful consideration since the requests to memory pass through the cache and can inherently impact the cache. We circumvent this problem by relying on NUMA machines with placement strategies designed to individually stress different memory subsystems. We use three system configurations as illustrated in Figure 1. They are designed to generate contention at different resources individually. The first configuration generates contention only on the cache, the second only on memory bandwidth and the third on both.

B. Co-profiling

The goal of stressing different memory subsystems is to understand the sensitivity of different Spark applications to contention at these subsystems. However, we are also interested in understanding the low level properties of both the Spark application and the co-runners that determine this sensitivity. We use performance monitoring units (PMU) to approximate this behaviour. PMUs are special registers in modern CPUs which collect low level hardware statistics of an application without any additional overhead. PMUs can only hold context either for a single process or multiple cores at a time. We circumvent this limitation by monitoring the Spark application and the co-runners in turns throughout the lifetime of the application execution. We also monitor the Spark application running in isolation in order to comparatively study the changes in the statistics under contention.

III. Sensitivity Analysis

In this section, we provide the list of metrics collected from the application and memory-subsystems. Due to the space limit, we only highlight the critical ones.

A. Application level

Spark provides a myriad of tuning parameters that can affect the application performance in ways that are often not obvious. An application life-cycle is typically composed of multiple jobs and stages, with each stage comprising multiple tasks. Task execution time can be further broken down into multiple phases: shuffle, garbage collection, computation and serialization/deserialization. From the memory interference point of view, we are particularly interested in understanding how an application life-cycle is affected by interference

and which part of the task execution is most impacted by interference. By knowing which execution phase is most impacted by interference, it is possible to tune the application to optimise the time spent on that phase and consequently mitigate the detrimental effects of interference. It also provides an overall understanding of how robust and flexible Spark is to interference.

To facilitate this, our tool decomposes an application life-cycle into multiple stages and aggregates information such as: average and maximum task execution time, stage completion time, and total number of tasks per host. The total number of tasks scheduled per host provides an insight into how Spark inherently deals with interference. Apart from aggregating stage information per host, the tool decomposes task execution time into phases and provides a comparative plot with and without interference to identify phases most impacted by interference. A key idea behind Spark is to enhance data locality by moving tasks to hosts where the data is located. In order to investigate what happens when hosts with local data are heavily interfered, our tool also gathers information on data locality of tasks.

B. Low level

The specific performance counters the tool monitors include: instructions, cycles, cache-references, cache-misses, LLC-prefetches, LLC-prefetch-misses and DTLB misses. They help understand both contentiousness of the co-running application and the sensitivity of the Spark application. For example, an application sensitive to cache contention will experience higher cache-misses when compared to its execution in isolation. Similarly, these counters when monitored for the co-runner help determine what properties of the co-runner create more contention on the memory subsystem. A co-runner with high cache-references is expected to be intensive on the cache. Once studied and validated, they can help schedule Spark applications and co-runners to be aware of performance interference.

C. Experiment Setup

All our experiments were conducted on our private cluster composed of three hosts each equipped with two Intel Xeon CPUs @ 3.00 GHz and 42GB of memory running Ubuntu 14.04. Each CPU has a 12 MB L3 cache and 12 hardware threads. We disable frequency scaling and pin Spark and the co-runners to physical cores to guarantee CPU isolation. Table I provides a summary of the applications and data sets used. We study representative workloads from machine learning, clustering and graph processing. The tool varies the data-locality, partition size, intensity of interference and the distribution of interference. Unless explicitly mentioned otherwise, the input data is replicated on all the hosts. Our analysis is preliminary, at a small scale and mainly serves as an indicator to demonstrate the capabilities of the tool.

TABLE I: Spark Applications

Benchmark	Category	Dataset	Description
Naive Bayes	Classification	KDD Cup 1999 [4]	5M network records
Decision Tree	Classification	KDD Cup 1999 [4]	5M network records
KMeans	Clustering	US Census 1990 [5]	2M instances
Triangle Counting	Graph processing	Live Journal [6]	5M nodes, 68M edges
Word Count	Minimal processing	Project Gutenberg [7]	4GB

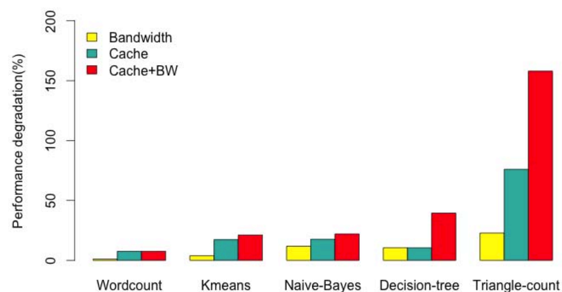


Fig. 2: Overall performance degradation for different Spark applications.

IV. Case Study

In this section, we present the preliminary evaluation results of analyzing the memory vulnerability of big-data applications.

A. Results

Dominant source of sensitivity: Figure 2 shows the overall performance degradation for different Spark applications with interference at different memory subsystems on all the hosts. Most of the applications degrade only by 25-35% except for triangle counting, which degrades by up to 150%. The results also show that all the applications are more sensitive to cache than bandwidth. This observation is consistent with previous studies [8] that show that Spark workloads use only a small portion of available memory bandwidth.

Distribution vs. Intensity: Figure 3 shows the performance degradation of triangle counting for varying intensities (number of co-runners) and distribution of interference (number of contented hosts). Naturally, the application degrades the most when both the intensity and the distribution of interference is highest. However, we find that even when the distribution of interference is small, performance degrades significantly. For example: with high interference on both cache and bandwidth only on one host, the application degrades by around 65%.

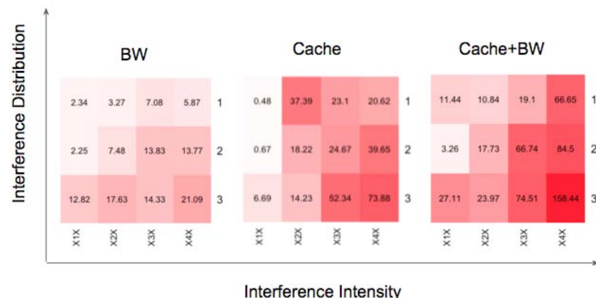


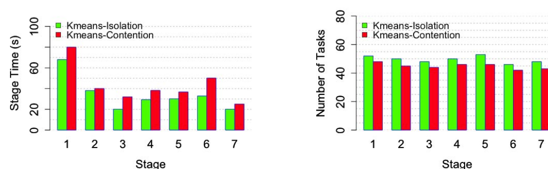
Fig. 3: Heatmap depicting the performance degradation of triangle counting for different interference intensities and distributions.

This impact is more prominent for applications that are highly sensitive to interference.

Application Breakdown: Figure 4a shows the application breakdown to stage execution time (we show only a subset of the data here) for a single host when running k-means. Of the three hosts used in the cluster, interference is generated on two hosts host 1 and host 2. From the figure, we can clearly see that the time taken to execute a stage increases in the presence of interference. Figure 4b shows the number of tasks (for a subset of the data) scheduled on host 1 with and without interference. In the presence of interference, the total number of tasks scheduled on host 1 is lower than the total number of tasks scheduled without interference. Spark schedules tasks on a host when the previous task on that host finishes its execution. In the presence of interference, the already scheduled tasks on the host take longer to finish their execution because of contention and as a result more tasks are scheduled on hosts that are not interfered. This shows that Spark scheduling is inherently adaptive to interference. Despite having a lower number of tasks, the overall stage execution time is longer in the presence of interference. However, the fact that Spark scheduling adapts to interference compensates for the increase in stage execution time.

Task Breakdown: Breaking down tasks further into phases, we observe that applications spending a significant amount of time in garbage collection or shuffling are heavily impacted by interference. The time taken for shuffling data is exacerbated in the presence of performance interference. Triangle counting is one such application that spends a significant amount of time in shuffling data, and consequently suffers greatly from contention.

Impact of Partition: It is generally recommended to chunk data into many partitions in order to spread out the work evenly across cores and achieve low latency in execution. We observe that tuning spark to have a high number of partitions also improves the resilience of the application to performance interference. With many partitions, data is readily available for Spark to schedule execution on hosts that have free resources. With few partitions, non-interfered hosts can have no data to work with while waiting for interfered hosts working on bigger chunks of data to finish which results in longer execution times.



(a) Stage completion time on host 1

(b) Total number of tasks scheduled on host 1

Fig. 4: Stage completion time and total number of tasks scheduled on host 1 when running K-means. Host 1 suffers from interference.

Impact of Data Locality: In order to test the impact of data locality, we concentrate all input data on a single host on which we generate high intensity of interference. Even under such a scenario, the performance of Spark application is not significantly degraded compared to an isolated run. We assume that this is because network is not the bottleneck. Any host can remotely fetch the data with little overhead even in the presence of interference.

V. Concluding Remarks

We present a methodology that can asses the vulnerability of memory subsystems of big-data applications, using a case study of five Spark applications. Our preliminary analysis on Spark shows that most applications tend to be more sensitive to cache than bandwidth and keeping the partition size small can improve the resilience to memory interference. As a part of the future work, we are interested in understanding how latency critical big-data applications such as real time streaming fare under performance interference and to see if their behaviour offers any predictability.

Acknowledgment

This work has been supported by the Swiss National Science Foundation (project 200021_141002).

References

[1] Govindan et.al. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of SOCC*, pages 22:1–22:14. ACM, 2011.

[2] Tang et.al. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of EXADAPT*, pages 12–21. ACM, 2011.

[3] Rameshan et.al. Stay-away, protecting sensitive applications from performance interference. In *Proceedings of Middleware*, pages 301–312. ACM, 2014.

[4] *KDD Cup 1999*, Last accessed: March 20, 2016. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[5] *US Census 1990*, Last accessed: March 20, 2016. https://archive.ics.uci.edu/ml/machine-learning-databases/census1990-mld/USCensus1990.html.

[6] *Live Journal Dataset*, Last accessed: March 20, 2016. https://snap.stanford.edu/data/soc-LiveJournal1.html.

[7] *Project Gutenberg Books*, Last accessed: March 20, 2016. https://www.gutenberg.org/.

[8] Jiang et.al. Understanding the behavior of in-memory computing workloads. In *Proceedings of IISWC*, pages 22–30. IEEE, 2014.