Reducing Monitoring Overhead in Virtualized Environments Through Feature Selection

Pedro F. Popiolek[1], **Karina S. Machado**[1], **Odorico M. Mendizabal**[1]

[1]Centro de Ciências Computacionais (C3) – Mestrado em Engenharia de Computação (PPGComp) – Grupo de Sistemas Digitais e Embarcados (GSDE)
Universidade Federal do Rio Grande - FURG
Caixa Postal 474 – 96.201-90 – Rio Grande – RS – Brazil

{p.f.popiolek, karina.machado, odoricomendizabal}@furg.br

Abstract. *Cloud computing has emerged as a cost-effective paradigm for hosting and delivering services. Cloud providers adopt server consolidation strategies to achieve efficient management of resources. A drawback is that applications running on the same host compete for physical resources. Such interference can affect the performance of applications. Performance monitors are useful tools to detect or even predict performance degradation. However, the monitoring itself can be a source of contention. In this paper, we analyze the influence of performance monitoring overhead in virtualized environments. Furthermore, as a mean to reduce contention for shared resources, we propose an approach to reduce the dimensionality of the performance feature space.*

1. Introduction

Cloud computing provides ubiquitous, convenient and on-demand network access to a shared pool of computing resources (e.g. servers, networks, storage, applications, and services) [Mell and Grance 2011]. From the business perspective, the pay-as-you-go model introduces flexible and cost-effective means to handle compute resource demands. From the infrastructure perspective, virtualization plays an important role on resources management. By multiplexing a bunch of Virtual Machines (VM) on the same physical machine, cloud platforms provide efficient management of resources by reducing underutilized hosts.

Typically, a physical server hosts several VMs through server consolidation [Zhang et al. 2010, Shirvani and Ghojoghi 2016]. However, individual applications running on the same host compete for physical resources like cache, memory and disk. Such cross-application interference can affect negatively the performance of applications running in the cloud [Popiolek and Mendizabal 2012, Rameshan 2016, McDougall and Anderson 2010, Pu et al. 2010]. A great challenge for cloud providers is to handle VM demands without overloading host machines. Once a physical machine runs out of its limits, graceful degradation and instability could affect the performance of hosted VMs.

Monitoring of resources is commonly used to watch systems health and identify contention. Hardware performance counters and operating system metrics can be used to detect interference [Fedorova et al. 2010]. However, depending on the amount of information collected, the monitoring overhead can be an issue. Specially in virtualized environments, where co-located applications compete for shared physical resources, the

monitoring itself can be a serious source of contention. Besides affecting system's performance, large amounts of data might limit users ability to identify performance bottlenecks. Hence techniques is needed to improve the select of performance counters by the users, consequently, will allow to reduce the monitoring overhead.

In this paper we propose the application of a Knowledge Discovery in Databases (KDD). This process is applied in order to identify a representative subset of performance counter to reduce the monitoring overhead generated to the system. According to [Han et al. 2011], in recent years data mining has attracted attention in academy, industry and in society as an approach to extract implicit, previously unknown, and potentially useful information from the huge amounts of data. Data mining is part of the KDD process that comprehends: data preprocessing; application of specific algorithms for extracting patterns from data (data mining) and post processing the obtained models and pattern evaluation [Fayyad et al. 1996, Han et al. 2011].

Thus, this paper has two main contributions. First, we analyze the monitoring overhead in virtualized environments. We experimentally evaluate how different patterns of applications can be affected by the monitoring overhead. Second, we propose a technique to reduce the monitoring overhead. The key idea is to identify and exclude redundant performance counters. Towards this end, counters are grouped hierarchically based on Pearson correlation coefficient. By selecting a representative counter of each group, we can reduce the number of variables to be observed.

Our results show a performance decrease caused by monitoring. Specially for I/O-intensive workloads, this overhead may cause a performance reduction around 30%. According to our results, the monitoring overhead was reduced by less than one third.

The rest of the paper is organized as follow. Section 2 surveys related work. Section 3 illustrates some causes of performance interference in virtualized environments. In Section 4 it is presented our approach to reduce monitoring overhead. Section 5 experimentally assesses the performance of the proposed approach, and Section 6 concludes this paper.

2. Related Work

As more applications run on clouds, it is becoming more important to leverage resources in a efficient way. While some approaches take into account application characteristics to identify which applications should be co-located, others analyze application load at runtime by reading data from performance monitors.

In [Jin et al. 2015] authors analyze cache access pattern in HPC applications running in clouds. They observe that applications with weak locality and large cache working sets profiles could be co-located with cache friendly applications in order to minimize cross-application interference. Mars et al. [Mars et al. 2011] present a methodology for prediction of performance degradation in virtualized environments. The approach is suitable for applications where overhead is mainly caused by contention for shared resources in the memory subsystem. Authors investigated the performance isolation of *latency-sensitive* tasks.

The aforementioned approaches rely on static approaches based on prior profiling of applications. In more general scenarios, where a fully characterization of applications

is unfeasible, the performance interference must be detected at runtime. In this sense, performance counters are largely used as a mean of watching systems performance.

Rameshan et al. [Rameshan et al. 2014] improve resource allocation by continuously learning the favorable and unfavorable behavior of co-execution. These informations are used to predict and prevent inefficient scheduling decisions. The approach proposed by [Zhang and Figueiredo 2006] also assists scheduling decisions based on performance monitoring of application-centric VMs. Snapshots of performance are taken at runtime and the collected data is processed in order to extract and classify relevant information about resource utilization. Their classification approach is based on featured selection algorithms, Principal Component Analysis, and the k-Nearest Neighbor classifier.

Our approach aims to reduce the monitoring overhead through featured selection. In [Zhang and Figueiredo 2006] authors propose a similar approach, but they limit the sample data to a small set of performance counters chosen in advance by an specialist. Our approach does not limit the number of counters and no previous knowledge about the set of performance counters under monitoring is required. Another feature selection approach is presented in [Yu and Liu 2004], but authors do not account for specific contention issues caused by virtualized environments. In [Shang et al. 2015], authors also evaluate correlation among performance counters, however, they focus in regression testing. In their work, they apply the same workload against different versions of an application and detect performance changes from one version to another.

3. Performance interference

The contention for shared resources is one of the main causes of performance degradation in computer systems. In virtualized infrastructures, such as cloud computing, the contention is even more perceptible due to the interference suffered by applications co-located in a same physical machine [Popiolek and Mendizabal 2012, Rameshan 2016].

The performance degradation may be caused by specific application patterns. Some applications are characterized by large cache resource consumption, which can create serious Shared Last Level Cache (SLLC) performance bottleneck [Alam et al. 2006]. Co-scheduled applications may induce interferences with cache access, which results in more cache misses and, as a consequence, performance degradation. VMs running in the same physical cores are subject to performance interference due to SLLC. As investigated in [Jin et al. 2015] cache contention breaks the inherent protection and isolation provided by virtualization, thus directly interferes performance of independent applications running in the same host.

Alves et al. [Alves and Drummond 2016] have demonstrated that the performance interference can also be influenced by the similarity of application's access burden. When applications co-located in the same host present a high level of access burden similarity for a given shared resource, they evenly compete for this resource which, in turn, leverages the level of interference suffered by these applications.

Moreover, the additional software layers responsible for emulation, virtualization of devices and the concurrency level among the guests result in performance cost [McDougall and Anderson 2010]. The scheduling performed by the virtual machine mon-

itor is also responsible by the decrease on performance in virtualized infrastructures. Usually, the scheduling policy assigns higher priority to partitioning of processor resources among active VMs than scheduling of I/O operations. For this reason, I/O-bound processes have worst performance than those CPU-bound [Pu et al. 2010].

As observed, there are many potential sources of contention. The performance damage depends on applications characteristics, scheduling and resource provisioning policies, or peculiarities of the virtualization technology. Thus, performance monitors are commonly used the investigate performance issues in these unpredictable scenarios. By gathering information about resources usage at runtime, monitors become an essential tool in data centers and cloud infrastructures. However, the collection of data performed by monitors introduces a non-negligible overhead to the system. Large software systems often collect thousands of performance counters during monitoring [Shang et al. 2015]. Next section presents an approach to reduce the interference caused by monitors.

4. An approach for reduction of monitoring overhead

Monitoring tools and application profilers use performance counters to collect data about resources usage. Performance counters track the usage of components such as processors, memory, or I/O devices. Then, by reading information from specific counters, one can detect bottlenecks and fine-tune system performance at run-time. Although it seems to be the normal activity of some counters are untouched by certain applications, understanding the impact caused by applications to a large set of counters would require the knowledge of a specialist. In addition, identifying the set of irrelevant counters manually would be time consuming and error-prone.

Thus, we propose a methodology to reduce monitoring overhead by applying the whole KDD process in order to select only the most relevant performance counters. The first step consists in collecting performance data. Then, we preprocess the obtained data by removing the counters with low variance measures. Next, we transform the attributes in a dissimilarity matrix. Finally, we apply a hierarchical clustering algorithm to identify which counters are similar, and reduce the set of counters by removing the redundant ones.

4.1. Step 1: collecting performance data

As a first step, it is necessary to select a set of counters among those available for monitoring. Usually operating systems and monitoring tools provide an API (Application Programming Interface) for users to register the performance counters of interest. The data acquired by these counters is normally stored in a log file. In our approach, the set of performance counters under monitoring is configured in advance. Unnecessary events, such as automatic updates and other scheduled system tasks, are disable in the system to not influence in the results.

Performance data is collected during an experiment execution, i.e., the system is stimulated by an application or benchmark in order to exercise systems resources while system usage data is measured through performance counters. To discard transient effects from warming-up time, a number of samples at the begin and end of the collected data is discarded.

Table 1 illustrates how the collected data is organized in the log file. The first column stores a timestamp indicating the time when data was collected and the successive columns shows the measured data for performance counters from 1 to n. Resource usage measured by each counter can be observed by lines 1 to m. Each sample $(i.j)$ is recorded in a specified time (i) and belongs a specified counter (j) following the distribution as Table 1 shows. To observe variations on a counter samples is necessary a workload that: stimulate this specific counter; and that changes its own behaviour.

Table 1. Disposition of collected performance data.

Timestamp	Counter 1	Counter 2	...	Counter n
time 1	sample 1.1	sample 1.2	...	sample 1.n
time 2	sample 2.1	sample 2.2	...	sample 2.n
...	...	...	...	...
time m	sample m.1	sample m.2	...	sample m.n

4.2. Step 2: preprocessing

The preprocessing phase aims to reduce the number of counters to be evaluated by our approach. After collecting performance counters data, we remove counters that are blank (corresponding to missing values) or that present zero variance. This step reduces the number of performance counters by removing those that were not influenced by the application behaviour. For example, supposing the performance counter *Current connections*, which measures the number of client connections established, is being monitored. Application workloads with no access to databases will not affect this counter, then it will be unmodified throughout the whole execution.

4.3. Step 3: dissimilarity matrix based on correlation

The reduced set of performance counters obtained in step 2 is used to create a dissimilarity matrix. A dissimilarity matrix stores a collection of proximities that are available for all pairs of n objects [Han et al. 2011]. In our approach, performance counters are the objects. Thus, each intersect of a row with a column has a value of dissimilarity based on Pearson correlation coefficient between the counter represented by the row and the counter represented by the column. Equation 1 shows the example of a dissimilarity matrix where $d(j_l, j_c)$ is the measured dissimilarity between the counter j_l and j_c.

$$\begin{bmatrix} 0 & & & \\ d(2,1) & 0 & & \\ d(3,1) & d(3,2) & 0 & \\ ... & ... & ... & \\ d(n,1) & d(n,2) & ... & 0 \end{bmatrix} \tag{1}$$

The Pearson correlation coefficient is a measure of the linear correlation between two objects. This coefficient ranges from -1 to 1. The extreme values indicate a perfect correlation: -1 for negative correlation; and 1 for positive correlation. The value 0 indicates no correlation. The farther from zero is the coefficient value, the higher the correlation: negative correlation when closer to -1; or positive, when closer to 1.

After calculating Pearson correlation coefficients *r* for the performance counters, the coefficients are transformed into a distance coefficients (*d*) to set the dissimilarity matrix. We apply this transformation using Equation 2. With this transformation the highest correlations, positive or negative, will get 0 as dissimilarity coefficient, and no correlations will get 1. The transformation is made in this manner because we are interested in evaluate correlations in the next step without worrying about the type of linear correlation.

$$d = \begin{cases} 1 - r & for \quad r \geqslant 0 \\ r + 1 & for \quad r < 0 \end{cases} \tag{2}$$

The Pearson correlation is a well known and widely used coefficient in scientific research to identify linear correlations [Taylor 1990]. Also, it is used to perform performance analysis in previous work [Shang et al. 2015].

4.4. Step 4: clustering similar counters

According to Han et al. [Han et al. 2011] clustering consists in the process of grouping the data into clusters where the objects within a cluster have high similarity to one another but are very dissimilar to objects in other clusters. The majority memory-base clustering algorithms consider as input data a specific data matrix. In this paper we are considering as input the dissimilarity matrix generated in Step 3.

We use a hierarchical clustering algorithm. This kind of cluster method creates a hierarchical decomposition of the given set of data objects that can be classified as agglomerative or divisive [Han et al. 2011]. We are using the hierarchical agglomerative algorithm called *complete linkage*.

In this algorithm each object of the dissimilarity matrix composes a group. The dissimilarity matrix is used to find the shortest distance between two groups. The distance between two groups is given by the distance of the most distant objects. The two groups found are grouped. The process is repeated iteratively analyzing the next shortest distance until form only one group. The process results a dendrogram illustrated by Figure 1. In our approach the objects in analysis are the performance counters.

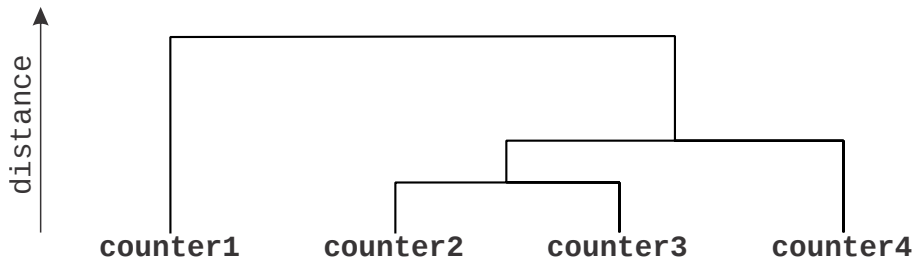

Figure 1. Dendrogram generated by a hierarchical clustering.

To define the number of groups desired is possible to specify a *distance* that the clustering process have to be cut, as shown in Figure 2. Counters connected by the branches below the defined cutoff compose a group. For instance, *counter2* and *counter3* compose the same group and counters *counter1* and *counter4* form separately other two different groups.

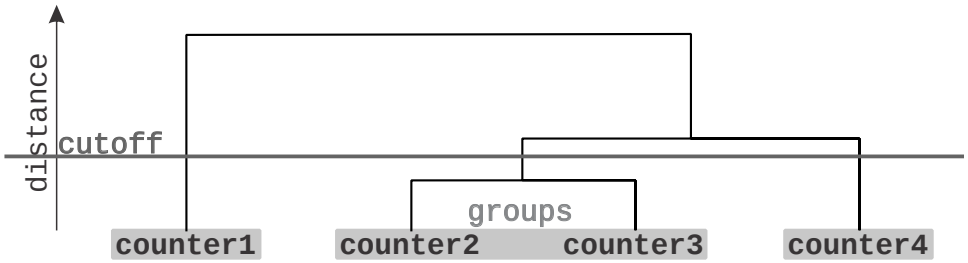

Figure 2. Cutting applied to a dendrogram.

Since the values on the matrix closer to 0 correspond to high correlation between the counters, and values closer to 1 indicate that a pair of counters have low similarity, the cutting applied to a dendrogram should be closer to 0 to form groups of counters with high correlation between them.

The cutting can be performed by many tree cut methods. Calinski-harabasz is a example of method successfully used by Shang et. al. [Shang et al. 2015] to perform hierarchical clustering of performance counters. But using this automatic method and other [Charrad et al. 2015], like Silhouette, there is no guarantee that the cutting will be made next to 0. Thus we decided to perform a static tree cut. In this work we apply the cutting at $distance = 0.1$ (i.e., $r = \pm 0.9$). According to [Taylor 1990], r coefficient describes a very high correlation when $r \geq 0.9$ or $r \leq -0.9$.

5. Evaluation

In this section, we explain our assessment goals and methodology, describe the workload and experiment environment, and present the results of our performance study.

5.1. Goals and methodology

Performance monitors are important tools to support decisions concerning resource provisioning on cloud computing. However, large amount of collected samples can affect performance of guest machines throughout their execution. Therefore, we wish to quantify the overhead caused by performance monitoring of application-centric VMs and evaluate the benefits of our approach. Our specific goals are described as follow:

- **Monitoring overhead:** The frequency and amount of data collected by performance monitors may influence systems performance. Higher frequency and large amount of data increase contention, which hurts performance. In virtualized environments, the contention effect becomes even more visible. We analyze the impact in application performance caused by monitoring overhead. We vary the number of virtual machines running in a physical host.
- **Workload impact:** The application behavior can exercise different computing resources, i.e., operating system or hardware components. We investigate how different patterns of applications are affected by the monitoring overhead. In particular, we evaluated CPU-intensive, memory-intensive, and I/O-intensive workloads.
- **Redundant counters suppression:** Information acquired by performance counters may present a strong correlation. Even with a suppression of some counters,

others can still capture the relevant information about resources usage. In our experiments we reduce the number of performance counters under monitoring by adopting our approach and evaluate the impact in the monitoring overhead.

- **Overhead reduction:** As a consequence of reduction on the number of performance counters under monitoring, it is expected a reduction on contention and competition for resources. We analyze the overhead decrease while monitoring the reduced set of counters selected by our approach.

5.2. Workload characterization

Applications for the most diverse purposes apply distinct load patterns against systems components. The impact on performance, though, can be difficult to predict. In order to provide controlled and reproducible workloads in our tests, we investigated the performance interference caused by a well-defined set of application workloads. To represent some application patterns, we implemented micro benchmarks to stretch specific systems resources. These benchmarks can be set to increase or decrease the workload intensity over the experiment life time. This is done by adding or removing work threads at specific instants of the execution. This feature enforces fluctuation on the load intensity throughout test execution.

Our benchmarks represent CPU-intensive, I/O-intensive and memory-intensive patterns. The CPU-intensive benchmark implements π calculation. It calculates the number π until its 100^{th} decimal place. The I/O-intensive benchmark implements random readings and random writings. These operations manipulate a pre-allocated 20GB of files and read or update blocks of 512 bytes per time. The memory-intensive benchmark implements similar operations over a pre-allocated 200MB-integer-array on memory. Both I/O and memory intensive benchmarks repeat the operations until the end of a pre-established execution time.

5.3. Environment and configuration

In our experiments the test environment is composed by 1 host machine with 4 cores (Intel Xeon E3-1240V5 processor), 16GB of RAM, 1TB of hard disk, operating system Windows Server 2012 R2 and VMware Workstation 12 Player as hypervisor. The hosted virtual machine are configured with 1 processors, 1 GB of RAM, 80 GB of virtual hard disk, and operating system Windows Server 2012 R2. The virtualization mode used was Intel VT-x/EPT and acceleration for binary translation was disabled.

Systems performance monitoring is performed by Perfmon (Performance Monitor) [Microsoft], a native tool available in Windows operating systems. This tool allows the selection of a set of performance counters, which are responsible for collecting information about processors, memory, disk, etc. In our experiments, data is collected every second and the number of performance counters under monitoring varies according to the experiment.

5.4. Monitoring overhead

We evaluate the monitoring overhead for different scenarios. The scenarios executed differ by the number of active VMs, varying from 1 up to 2 VMs running concurrently, and by the kind of workload. The modeled workload profiles represent CPU-intensive,

I/O-intensive (disk reads, writes and reads/writes) and memory-intensive (reads/writes) processes. For all scenarios, each active VM was exposed to exactly the same workload.[1]

In our experiments, a single thread is responsible for measuring the application throughput. The time interval between throughput sampling is 1 second. To analyze monitoring overhead, we have measured the application throughput in two different setups. In the first case, to avoid external contention, only the application benchmark is executing. Then, to observe monitoring overhead, we run the application together with the monitoring tool. When monitoring is running, the Perfmon tool is configured to collect information from 9,108 performance counters.

The overhead caused by monitoring calculation gives a ratio between the throughput observed when monitoring is disabled and when it is enabled. The overhead returns the percentage of operations that could be executed whether monitoring was disabled. Considering the average operations per second performed with monitoring enabled (OME) and the average operations per second performed with monitoring disabled (OMD), the overhead is given by Equation 3.

$$Overhead(\%) = 100 - \left(\frac{OME \times 100}{OMD}\right) \tag{3}$$

Figure 3 shows the number of reads commands in a single VM for executions of the I/O-intensive benchmark set to perform read-only operations. In this scenario only one VM is executing, i.e., there is no other VMs allocated to the host machine. The continuous line shows the throughput when monitoring is disabled and the dashed line when it is enabled. As can be observed, the number of read commands is consistently lower when the monitor is executing. The average throughput when performance monitor is disabled is 73.17 operations per second, while the average throughput when monitor is enabled is 51.46 operations per second. The monitoring overhead is 29.67%, which means the performance of application is almost 30% worst due to the monitor influence.

Figure 4 depicts the monitoring overhead for applications running on multiple VMs. In our experiments, the infrastructure was configured as: just a single VM, or two VMs running in a host. The bar chart indicates the application patterns, which are represented by memory-intensive, CPU-intensive and disk-intensive workloads. The disk benchmark was set up to perform read-only operations (Disk R), write-only operations (Disk W), and a mix with 50% of read and 50% of write operations (Disk R/W). The clustered bars labeled as "1 VM" and "2 VMs" shows the monitoring overhead for executions with one guest VM and two guest VMs, respectively. With 1 and 2 VMs executing, the monitoring overhead for CPU and memory-intense workloads is between 4% and 7%. These benchmarks are the less affected, since monitoring activity is mostly disk-intensive. The impact of monitoring becomes more noticeable when application also competes for disk, which is the case of disk-intensive workloads. As observed, for read-only operations the monitoring overhead accounts for up to 30%.

[1]Experiments with more than two VMs presented a severe performance degradation when the whole set of counters where being monitored, so we decided not to include them in our analysis.

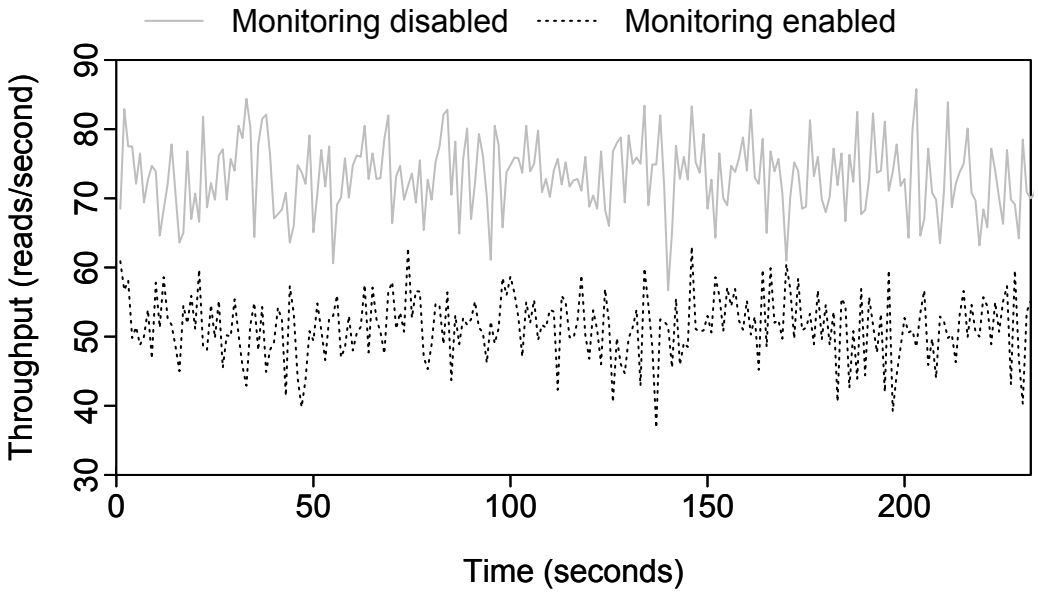

Figure 3. Throughput of read-intensive workload in a single VM.

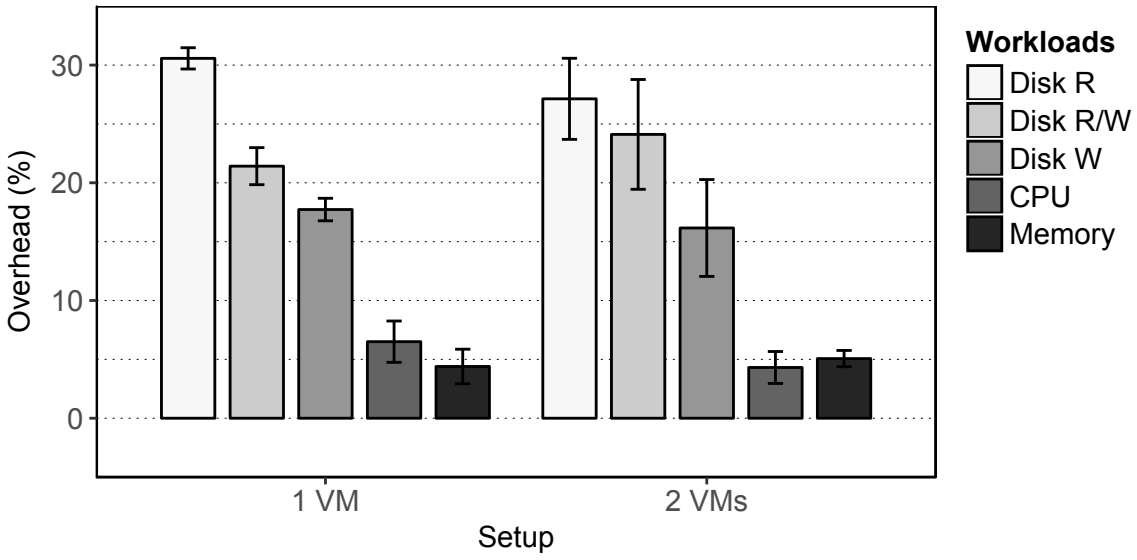

Figure 4. Monitoring overhead using full performance counter set according to the number of virtual machines.

5.5. Performance counters correlation

The main goal of this paper is to reduce monitoring overhead without affecting the monitor accuracy. Our approach returns a representative subset of counters based on feature selection of performance counters for each application pattern. By applying the steps defined in Section 4 over the data acquired by experiments from Section 5.4, we observe an expressive reduction in the number of relevant performance counters. From a total of 9,108 performance counters, we have selected 124 counters for the CPU-intensive benchmark; 384 for the read-intensive benchmark; 316 for the write-intensive benchmark; 384 for read/write intensive benchmark; and 376 for memory-intensive benchmark.

With less performance counters under monitoring, it is expected a reduction on resources usage and contention. Monitor will have less influence in scheduling and interruption time (e.g. CPU time allocation and logging information). On the one hand, with a reduced set of counters, there is less data about the system to collect and store. On the other hand, less information would give a less accurate view of the system health.

Our approach reduces only counters that have a high correlation with others. Therefore, the ability to evaluate systems performance is not prejudiced. For example, performance counters *% Processor Time* and *% Idle Time* measure the percentage of CPU usage and idleness, respectively. Consequently, as one increases, the other decreases proportionally. For that reason, only one of them need be monitored. Analogously to this example, other pairs or groups of counters describe complementary information, then some of them can be omitted. Our approach captures these correlations among counters. Notice that figuring out these correlations manually for a large number of counters would be impractical.

5.6. Overhead reduction

After reducing the the performance feature space by excluding redundant counters, we reproduce the same test scenarios described in Section 5.4 and evaluate the monitoring overhead.

Figure 5 shows the number of read commands performed by a single VM running at the host machine. This scenario replays the workload applied in Figure 3. As can be observed in the graph, the overhead caused by the performance monitor is almost imperceptible. The average throughput when monitor is disabled is 73.17 operations per second. When monitoring is enabled the throughput reaches 67.08 operations per second. This represents an important reduction of monitoring overhead, since the application performance were around 51.46 operations per second before feature selection.

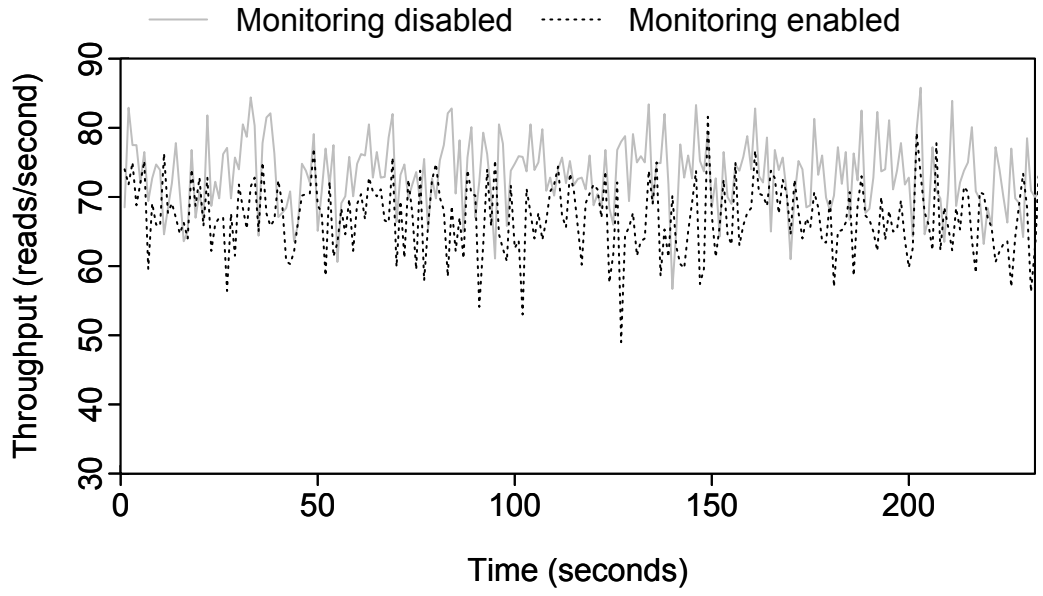

Figure 5. Throughput of read-intensive workload in a single VM after performance counters selection.

Figure 6 compares the monitoring overhead caused by the measurement of the

whole set of counters and the reduced set of counters. Solid bars represent the overhead caused by the original set of counters and dashed bars shows the overhead caused by the reduced set of counters. As observed, the performance counter set nominated by our approach consistently reduces the monitoring overhead for all workload patterns. CPU and memory-intensive workloads present a negligible overhead when the subset of counters is configured. The highest overhead observed by the optimized monitor is perceptible in read-only, disk-intensive workload, approaching 10%. However, this overhead corresponds to only one third of the overhead measured in the equivalent scenario with the whole set of counters enabled.

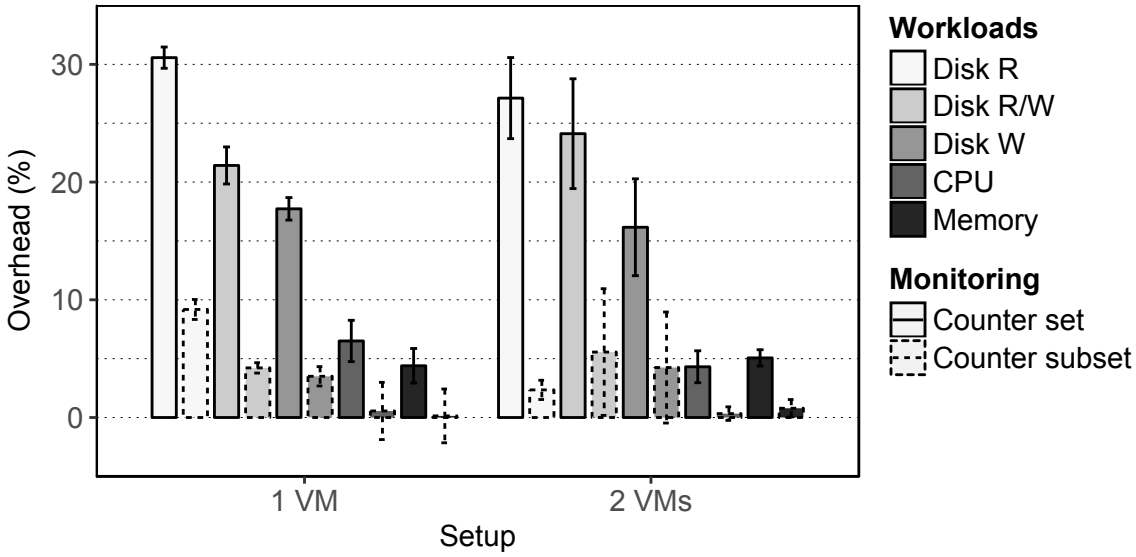

Figure 6. Monitoring overhead contrast between the use of full performance counter set and reduced performance counter set according to the number of virtual machines.

6. Conclusion

With a growing number of applications running on cloud, it is becoming more important to leverage resources in a efficient way. Performance monitors, through information collect from performance counters, are largely used as a mean of watching systems performance. They are valuable tools for supporting loading balance, migration and resource management decisions. However, the monitoring itself can hurt performance, specially in virtualized environments.

This paper analyzes the performance overhead caused by system monitoring in virtualized infrastructure. Different application patterns were evaluated and experimental results reveal significant resource contention introduced by monitoring activity. I/O-intensive applications are the most affected, especially those with read-only operations.

We propose an approach capable to reduce monitoring overhead without affecting the monitor accuracy. Our approach uses data mining strategies to select only the most relevant performance counters for different application workloads. We apply hierarchical cluster analysis to find out a relevant group of performance counters. By using the selected set of performance counters, the monitoring overhead is drastically reduced. In some

cases, the overhead can be one third lower than the one observed with the whole set of counters.

References

Alam, S. R., Barrett, R. F., Kuehn, J. A., Roth, P. C., and Vetter, J. S. (2006). Characterization of scientific workloads on systems with multi-core processors. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 225–236. IEEE.

Alves, M. M. and Drummond, L. M. d. A. (2016). A quantitative model for predicting cross-application interference in virtual environments. *arXiv preprint arXiv:1610.04309*.

Charrad, M., Ghazzali, N., Boiteau, V., and Niknafs, A. (2015). *Determining the Best Number of Clusters in a Data Set*. https://cran.r-project.org/web/packages/NbClust/NbClust.pdf, 3,0 edition.

Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. (1996). The kdd process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34.

Fedorova, A., Blagodurov, S., and Zhuravlev, S. (2010). Managing contention for shared resources on multicore processors. *Communications of the ACM*, 53(2):49–57.

Han, J., Kamber, M., and Pei, J. (2011). *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition.

Jin, H., Qin, H., Wu, S., and Guo, X. (2015). Ccap: a cache contention-aware virtual machine placement approach for hpc cloud. *International Journal of Parallel Programming*, 43(3):403–420.

Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L. (2011). Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM.

McDougall, R. and Anderson, J. (2010). Virtualization performance: perspectives and challenges ahead. *ACM SIGOPS Operating Systems Review*, 44(4):40–56.

Mell, P. and Grance, T. (2011). The nist definition of cloud computing. *Retrieved from http://csrc. nist. gov/publications/nistpubs/800-145/SP800-145.pdf*.

Microsoft. Windows reliability and performance monitor. https://technet.microsoft.com/en-us/library/cc749249(v=ws.10).aspx. Last accessed: Jun-2017.

Popiolek, P. and Mendizabal, O. (2012). Monitoring and analysis of performance impact in virtualized environments. *Journal of Applied Computing Research*, 2(2):75–82.

Pu, X., Liu, L., Mei, Y., Sivathanu, S., Koh, Y., and Pu, C. (2010). Understanding performance interference of i/o workload in virtualized cloud environments. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 51–58. IEEE.

Rameshan, N. (2016). On the role of performance interference in consolidated environments. In *IEEE/USENIX International Conference on Autonomic Computing (ICAC)*. KTH Royal Institute of Technology.

Rameshan, N., Navarro, L., Monte, E., and Vlassov, V. (2014). Stay-away, protecting sensitive applications from performance interference. In *Proceedings of the 15th International Middleware Conference*, pages 301–312. ACM.

Shang, W., Hassan, A. E., Nasser, M., and Flora, P. (2015). Automated detection of performance regressions using regression models on clustered performance counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 15–26. ACM.

Shirvani, M. H. and Ghojoghi, A. (2016). Server consolidation schemes in cloud computing environment: a review. *European Journal of Engineering Research and Science*, 1(3):18–24.

Taylor, R. (1990). Interpretation of the correlation coefficient: a basic review. *Journal of diagnostic medical sonography*, 6(1):35–39.

Yu, L. and Liu, H. (2004). Efficient feature selection via analysis of relevance and redundancy. *Journal of machine learning research*, 5(Oct):1205–1224.

Zhang, J. and Figueiredo, R. J. (2006). Application classification through monitoring and learning of resource consumption patterns. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE.

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18.