Estimating the Impact of External Interference on Application Performance

Aamer Shah[1], Matthias Müller[1], and Felix Wolf[2(✉)]

[1] IT Center, RWTH Aachen University, Aachen, Germany
{shah,mueller}@itc.rwth-aachen.de
[2] Laboratory for Parallel Programming, TU Darmstadt, Darmstadt, Germany
wolf@cs.tu-darmstadt.de

Abstract. The wall-clock execution time of applications on HPC clusters is commonly subject to run-to-run variation, often caused by external interference from concurrently running jobs. Because of the irregularity of this interference from the perspective of the affected job, performance analysts do not consider it an intrinsic part of application execution, which is why they wish to factor it out when measuring execution time. However, if chances are high enough that at least one interference event strikes while the job is running, merely repeating runs several times and picking the fastest run does not guarantee a measurement free of external influence. In this paper, we present a novel approach to estimate the impact of sporadic and high-impact interference on bulk-synchronous MPI applications. An evaluation with several realistic benchmarks shows that the impact of interference can be estimated already based on a single run.

1 Introduction

On many HPC systems, the execution time of applications varies considerably between runs, which makes performance measurements hard to reproduce and challenges their validity. Possible sources of variation include operating system jitter, different process-to-node mappings, or contention on shared resources. While modern operating systems reduced their noise footprint [16], contention on heavily loaded centralized file systems and communication interconnects, such as torus and dragonfly networks, are still contributing to performance variation [3,21]. Because such external interference occurs randomly, benchmarking has become complicated.

Usually, performance analysts prefer measurements that are as close as possible to an application's *intrinsic* behavior, that is, without external influence beyond their control. To achieve this on a system with strong performance interference among jobs, one could take multiple measurements and pick the run with the shortest execution time or the average or median if a certain degree of interference is considered natural. No matter how, this strategy is both expensive and unreliable because neither may the minimum be free of interference nor the

M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 46–58, 2018.
https://doi.org/10.1007/978-3-319-96983-1_4

average or median representative. After all, the system load also changes along macroscopic time scales (e.g., daytime or season).

To help performance analysts decide how much they can "trust" their benchmarking results and whether they need to repeat measurements, we present a novel approach to estimate the impact of external interference on the execution time of a common class of MPI applications. As a distinctive feature, our method can deliver such an estimate with negligible overhead based on a single run. Moreover, it is agnostic to the source of interference. Instead, it exploits the properties of bulk-synchronous MPI applications that perform frequent global all-to-all operations. Such applications not only make up a significant portion of HPC workload (almost two-thirds of unique benchmarks in the SPEC MPI suite fall in this category), they are also most sensitive to external interference [1,7,10].

The remainder of the paper is structured as follows. While Sect. 2 provides the details of our approach, Sect. 3 demonstrates the accuracy of our estimates in a series of experiments. After presenting related work in Sect. 4, we review our results in Sect. 5.

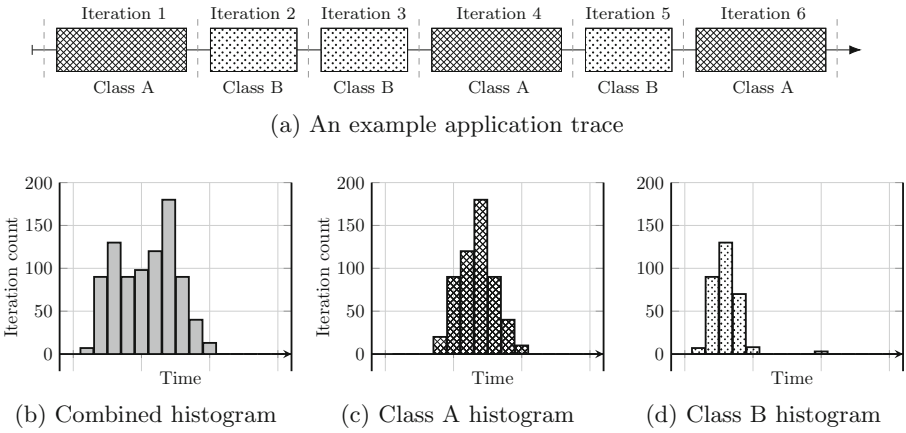
2 Approach

Most HPC applications are iterative in nature. After a brief initialization, they go through different phases that are repeated over and over. Similar phases have similar execution times unless a phase instance is struck by external interference. The stronger the impact, the greater the elongation of the execution time.

Figure 1a shows a trace snippet of a typical HPC application. The application performs several iterations, whose execution times are, however, not uniform. The execution-time histogram in Fig. 1 illustrates two sources of variation – one intrinsic and one extrinsic. Intrinsic variation arises from programmatic differences because, for example, some iterations may calculate some extra physics every once in a while or store checkpoints. The example shows two classes of iterations, A and B, distinguished by their programmatic characteristics and visible as two peaks in Fig. 1b. The variation that remains after separating these two classes, as shown in Fig. 1c and 1d, is extrinsic and the result of noise such as interference from other jobs that happen to run at the same time.

The key idea of our approach is to divide the execution of a program into segments and classify them according to their intrinsic characteristics. In a noise-free environment, segments within each class are then expected to consume the same amount of time. Conversely, variations that occur within each class are likely caused by noise. Because execution time is subject to such noise, we have found hardware and software counters that reflect computation, communication, and file I/O features to be suitable metrics for our programmatic classification of execution segments.

To identify segments, we take advantage of the bulk-synchronous nature of many HPC applications, specifically we exploit periodic (blocking) all-to-all communication. Although this practically restricts our method to such applications, we claim that we can still cover major portions of today's HPC workloads. After



(a) An example application trace

(b) Combined histogram (c) Class A histogram (d) Class B histogram

Fig. 1. Application iterations and their histograms.

all, this is not an uncommon feature. For example, almost two-thirds of unique benchmarks in the SPEC MPI suite fall into this category. At the same time, applications with frequent all-to-all communication suffer more than others from external interference because every delay of a process will likely induce waiting time in all others.

We use global all-to-all communication operations as a boundary between execution segments. These might not exactly match programmatically specified iterations, but are expected to divide the execution into repeatedly executed pieces. For example, an all-to-all operation will likely appear at least once in every iteration of the core loop. Furthermore, such all-to-all operations constitute global synchronization points among processes. Although the MPI standard does not explicitly require it, the nature of all-to-all operations implies it. This makes the execution segments between them independent with respect to the propagation of wait states that occur in response to external interference. A wait state whose root cause lies within a segment will not propagate across a global synchronization point [4]. For applications using non-blocking collectives, the wait operation of the collective call could be used as a boundary indicator, while for non-bulk-synchronous applications recurring MPI calls may serve this purpose. However, in both cases, adjacent execution segments may not be fully independent, with wait states and interference-induced delays potentially propagating across segment boundaries. We therefore concentrate on blocking collectives in this study and consider the remaining cases as future work.

Profiling Methodology. To classify segments, we count computation, communication, and file I/O operations or volumes per segment and process using LWM^2 [18], a low-overhead profiler, which leverages the PMPI interface to find segment boundaries and collect metrics related to MPI. The specific metrics we capture are discussed further below. At the end of each all-to-all collective call, the profiler stores information pertaining to the completed segment in memory.

To reduce storage requirements, the values for each metric are quantized into 256 unique bins. When the number of unique values exceeds the number of bins, the two bins with the least distance between them are merged. Instead of actual values, a segment profile stores the indices of the corresponding bins. Whenever bins are merged, the indices in the segment profiles are updated accordingly. After the program has ended, we merge per-process bins into 512 unique program-wide bins. To keep computation diversity among segments manageable, a segment is always represented by the median of the computation feature metric across processes. For communication and file I/O, such aggregation is only performed if the diversity among segments exceeds a threshold.

Grouping Segments Based on Computation Features. To classify segments, we first compare them in terms of the amount of computation they are supposed to complete. To measure the amount of work, we count the number of floating-point instructions using hardware counters. When the floating-point counter is not available, as on some generations of modern processors, we use the total number of completed instructions as a proxy. To shield them from noise, we only count them outside communication or I/O operations. While the captured values are still perturbed by OS jitter, we have found floating-point operations to be most stable. The total instruction count shows still less than 1% variability.

We establish similarity among segments by clustering them based on the above instruction counts as features. As the duration of segments in an application can vary widely, the possible range of feature values can be quite large. Furthermore, OS jitter and inaccuracies introduced when reading and storing hardware counters [6] cause variation among hardware-counter values from similar segments. Therefore, the most appropriate clustering algorithm for our task needs to handle a large range of values, and at the same time be tolerant to variations inside a cluster.

Common clustering algorithms such as k-means require the number of clusters to be known *a priori*. If such information is not available, such algorithms are executed for a range of cluster counts and an internal cluster criterion, such as the Calinski and Harabasz (CH) criterion, is applied to identify the most appropriate number of clusters. Even for a particular number of clusters, these clustering algorithms require several iterations to find the optimum centroids. These factors result in algorithms that, overall, are complex to implement and can take a significant amount of time for large numbers of data points.

Clustering with Relative Distance. Density-based algorithms such as DBScan seem to present an alternative. They can identify the appropriate number of clusters in a single pass. Such algorithms use a distance threshold to split the data points into clusters. However, relying on a fixed distance for a large range of values results in either merging distinct clusters with lower values if the threshold is too large, or splitting a single cluster with a modest range of higher values into multiple clusters if the threshold is too small.

To overcome these difficulties, we designed a simple clustering algorithm that can identify clusters in one-dimensional data even with a large value range in

a single pass. The algorithm requires the data type to have a total order and a threshold for the maximum *relative* distance between any two data points in a cluster. We define the relative distance between two points as their distance divided by the smaller of the distances of the two points from the origin. As the algorithm relies on relative distance, it can identify clusters with a modest degree of internal variance both at the lower and higher end of the value range. Our algorithm first sorts all the values in ascending order and then assigns the smallest element to the first cluster. After that, it iterates through the remaining sequence and, at each step, picks the value at position i from the sorted list that was assigned to a cluster in the previous step and determines the relative distance to the next value at $i + 1$. If the relative distance is less than the threshold, the value at $i + 1$ is placed in the same cluster as the value at i. Otherwise, a new cluster is created for the value at $i + 1$.

Using SPEC MPI benchmarks, we compared our new algorithm with k-means and an expectation-maximization (EM) algorithm that assumes the data to exhibit a mixed Gaussian distribution. Specifically, we analyzed the mean normalized standard deviation of the created clusters and the percentage of segments that ended up in clusters of less than five elements, which is the minimum size below which clusters are not considered for interference estimation. K-means identified tightly fitting clusters but left a larger portion of segments unclustered (up to 8%). EM, on the other hand, clustered almost all segments, but created clusters of high internal variance. We tried our new algorithm with several relative-distance thresholds, including 0.2, 0.1, and 0.05. With a relative-distance threshold of 0.1, the threshold we use in the remainder of this study, our customized algorithm identified clusters with slightly higher variance than k-means, but left only half the number of segments unclustered.

Grouping Segments Based on Communication and File I/O Features. As communication and I/O features of a segment we consider the number and accumulated volume of communication and I/O operations, including the number of point-to-point send/receive calls broken down by their blocking semantics, the number of collective calls broken down by their number of senders vs. recipients, and the number of bytes sent or received through them. Similarly, as file-I/O features we capture the number of open/close operations, the number of read/write operations and the accumulated number of bytes read or written. Since there is no clear relationship between these metrics and the execution time of a segment, we consider the corresponding values as nominal data. For example, a segment may run longer than another segment, although its number of sends is smaller. At the same time, these metrics are fairly stable and usually not subject to any jitter. Thus, we consider all segments that share the same unique combination of communication and file I/O metrics a separate group.

Estimating Interference. We estimate the impact of interference based on the segment profile of a single run. First, we cluster the segments according to their computation features, as described earlier. After that we split each cluster into

groups according to the communication and file-I/O features of its elements. The segments in each of the resulting groups are assumed to exhibit similar behavioral characteristics and consume about the same intrinsic execution time.

Any segment in a group that has a significantly higher execution time is considered to be affected by interference. More precisely, we classify a segment as interfered if its execution time is four MAD greater than the median of the group, with MAD (Median Absolute Deviation) being $\texttt{MAD} = \texttt{median}(|X_i - \texttt{median}(X)|)$. Median and MAD are known for their robustness to variability. The threshold of four MAD greater than median gives a confidence interval of more than 99.5%. The impact of interference on a segment is estimated as the portion of execution time of the segment in excess of the threshold. Adding the interference impact computed for all segments yields the interference impact for the entire program and is provided as a percentage of the (interfered) execution time.

Separating Instantaneous Interference from Continuous Interference. Execution time variation can arise from either high-frequency but usually low-impact interference such as certain types of OS jitter or from low-frequency but often high-impact interference such as sudden I/O contention. We call the former kind continuous interference, and the latter kind instantaneous interference. Continuous interference affects almost all segments of a profile, and as a result also affects the median in a group. In contrast, instantaneous interference only affects selected segments, and the median remains largely unaffected. While both kinds of interference prolong execution time, instantaneous interference is more likely to create undesirable artifacts in performance measurements a performance analyst may wish to remove. In contrast, continuous interference is often seen as an unavoidable evil one has to live with on a given system. Our approach only reports instantaneous interference. The median displacement caused by continuous interference ensures that it leaves no imprint on our estimates.

Tool Workflow. LWM^2 profiles the target application during execution, capturing the required metrics separately for each segment. At the end of the execution, LWM^2 writes a segmented profile to disk. Later, the profile is subjected to automatic interference estimation in Matlab. First, we classify the segments into different groups based on their features. Later, we estimate the impact of interference first for each segment group, and then aggregate the results for the whole application.

3 Evaluation

To evaluate our approach, we use the following benchmarks: (i) those seven codes from the SPEC MPI 2007 suite V2.0 that are bulk-synchronous according to our definition and that have a large data set available; (ii) Sweep3D, a time-independent 3D neutron transport simulation; and (iii) HACC, an application that simulates the formation of collision-less fluids and whose regular

checkpointing behavior makes it a popular I/O benchmark. We test our method both in a controlled environment with artificially injected interference, and on a production system with real interference.

Experimental Setup. Because of its low OS jitter, we chose JUQUEEN, an IBM BlueGene/Q system, as our controlled environment. Each of its 28,672 compute nodes consist of a 16 core IBM PowerPC®A2 processor and 16 GB of memory. JUQUEEN has a 5D Torus communication interconnect and minimizes network interference by making node boards with 512 cores the smallest allocation unit. Since its GPFS file system is shared, JUQUEEN cannot be considered controlled for I/O intensive workloads though, which, however, among our benchmarks only affects HACC. For our tests under production conditions, we use Hazel Hen, a Cray XC40 system with 7712 compute nodes, each of them featuring two 12-core Intel Haswell E5-2680v3 processors and 128 GB of memory. Applications running on Hazel Hen are known to experience significant run-to-run variation, majorly due to cache misses in the Aries chip under heavy network load from multiple applications [9].

Evaluation Methodology. With the exception of the file system, our controlled environment is without any significant natural interference. This is why the runtime of a job is usually close to its intrinsic execution time, providing us with a ground truth for interference-free execution. To test our method, we inject artificial interference into application runs using a tool called intM (interference Modeler), which we have developed for this purpose. intM sits as an interposition wrapper between an application and the runtime, and mimics network and file I/O interference by introducing delays in function calls. intM supports interference injection in MPI communication and I/O functions, as well as in POSIX I/O. The interference added to the regular execution time follows a Gaussian distribution, with configurable mean and standard deviation. The probability of when an interference event strikes a communication or file I/O operation is also configurable.

Specifically, we inject gradually increasing interference into multiple runs of a benchmark. Figure 2 shows such runs for the SPEC MPI benchmark tera_tf as an example. We compare the *estimated* with the *measured* impact of interference on each run. Measured interference is the execution-time difference between a run and the fastest run in percent of the (interfered) runtime. Estimated interference is calculated individually for each run as percentage of its runtime using our approach without considering any other run. To clean the measured interference from effects of continuous interference and other influences that are largely constant across the entire duration of a run but may vary between runs, such as different process-to-node mappings, we reduce the measured interference by the amount of time the medians are displaced. We observe the median displacement during clustering, and attribute it to continuous interference.

We categorize the impact of interference into the classes low, medium, and high, as shown in Fig. 2. A low-interference run is perturbed to a negligible degree

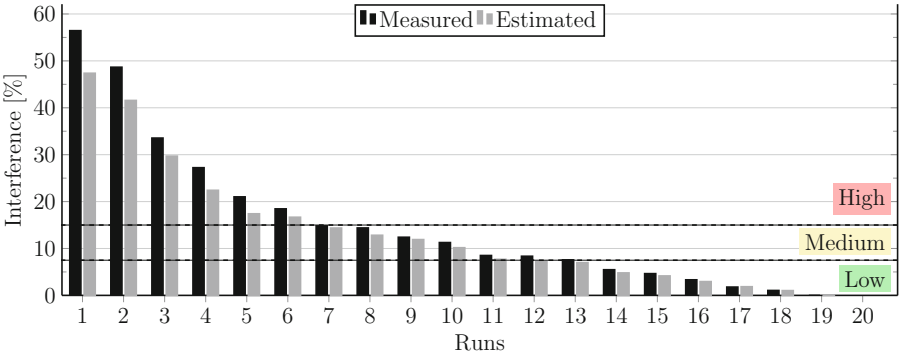
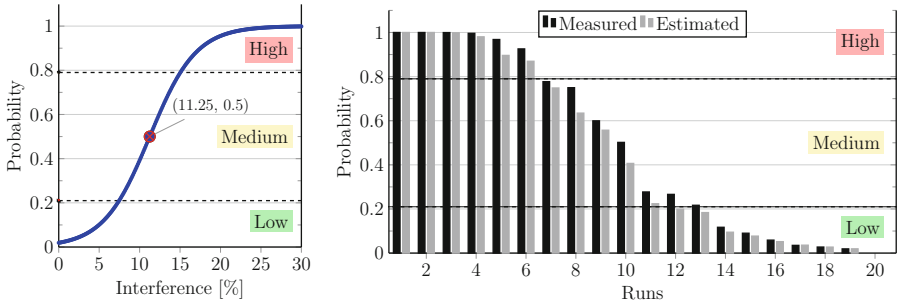


Fig. 2. Multiple runs of tera_tf on JUQUEEN, with measured and estimated interference classified as low, medium, or high. Runs are sorted by execution time in descending order.



(a) Logistic function (b) Highly interfered run probabilities

Fig. 3. Logistic function and the highly interfered run probabilities, when the function is applied on the tera_tf runs.

and can be used for performance analysis, whereas a high-interference run is heavily perturbed and should be discarded. The medium category is between these extremes: It might be worthwhile to invest in a new performance measurement, while, at the same time, the run can be used to gauge performance at large. Using the analogy of a traffic light, low means green light for performance analysis, medium means yellow light, and high red light. We have set the threshold for low interference to below 7.5%, for high interference to above 15%, and classify everything in-between as medium. While such categorization is useful to distinguish runs in practice, accuracy evaluation via *hard* classification into these three categories can run into pitfalls. For example, even if the difference between measured and estimated interference of a run is small, the two interference values can still fall into different classes, as it happened for runs 12 and 13 in Fig. 2. An alternative way of aggregating our results is calculating the percentage-point difference between measured and estimated interference. The

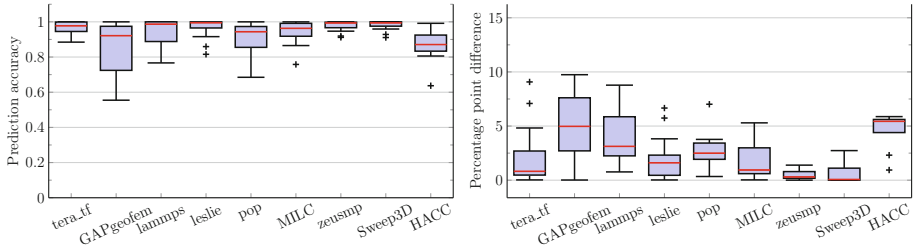
downside of this approach is that for highly interfered runs, the percentage-point difference is not that critical as long as both agree on the judgment that the run is highly interfered. Runs 1 and 2 in the figure are such cases.

Based on the intuition that the impact of interference is a measure of a run's suitability for performance analysis, we use a logistic function as a *soft* classifier to convert the magnitude of interference into the probability of a run being highly interfered. Using soft classification, the probability that a run previously categorized as low is actually highly interfered should be close to zero, while for runs categorized as high it should be near one. Similarly, at the mid-point of the medium category, the probability should be exactly 0.5. Figure 3a shows a logistic function that we have designed for this purpose, while Fig. 3b shows the corresponding probabilities for the tera_tf runs.

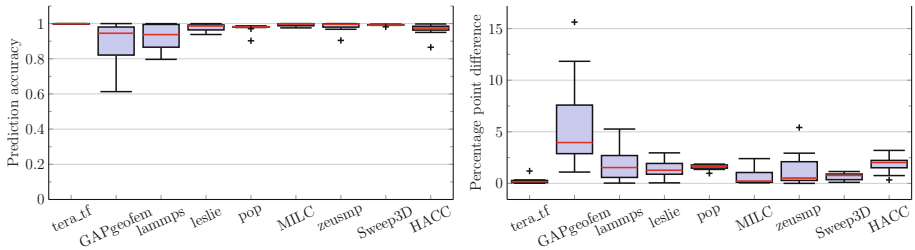
Formally, a logistic function is an "S" shaped function that maps values from $(-\infty, \infty)$ onto $(0, L)$. It is defined as $f(x) = \frac{L}{1+\exp^{-k(x-x_0)}}$, where k is the steepness, x_0 is the inflection point, and L is the maximum. As explained before, we define the inflection point, x_0, to be 11.25, the mid-point of the medium class. Similarly, setting the maximum value, L, to 1, and steepness, k to 0.35, the probabilities at interference magnitudes of 7.5%, 11.25%, and 15% are 0.21, 0.5, and 0.79, respectively.

Using this logistic function, we derive probabilities for measured and estimated interference for each run of a benchmark. The difference between the two probabilities is the inaccuracy of interference prediction, and its compliment is the accuracy. We determine the accuracy of our approach for all the runs of each benchmark and draw the results as boxplots (Fig. 4). As the logistic function in Fig. 3a shows, an accuracy of less than 0.5 means a significant deviation between measured and estimated interferences. To also give a more direct impression of the results, we complement probability differences with boxplots of the percentage-point difference between measured and estimated interference.

Results. On JUQUEEN, our controlled environment, each benchmark was executed at least 15 times with a gradually increasing amount of artificial interference injected. Figure 2 shows the series for tera_tf as an example. The interference was adjusted in such a way that multiple runs were produced for each interference class. We executed each benchmark on 256 nodes, with 4 processes running on each node. Figure 4a presents on the left how accurately we predict the interference probabilities and on the right the percentage-point difference between measured and estimated interference. Except for GAPgeofem, the median accuracy for all the benchmarks on JUQUEEN is above 0.9. Similarly, for most benchmarks, the minimum accuracy is above 0.8. This shows that in most cases estimated and measured interference leads to the same conclusion. That the accuracy of our predictions for certain runs of GAPgeofem was low can be attributed to its high collective-call rate of around 300 Hz. At such a high frequency, large numbers of small execution segments are created, easily leading to measurement artifacts that disturb our analysis.



(a) JUQUEEN



(b) Hazel Hen

Fig. 4. Prediction accuracy as difference of soft classification probability (left) and percentage-point difference (right) between measured and estimated interference.

Because of its low run-to-run variation, we also used JUQUEEN to evaluate the overhead of our profiler. Using the same set of benchmarks, we executed each benchmark on 128 nodes, with 4 processes running on each node. For each benchmark, we executed two series of experiments, one instrumented and one uninstrumented. To avoid bias caused by daytime differences, we interleaved the execution of the two series, alternating between the instrumented and the uninstrumented version. Each series consisted of nine experiments. Measured by comparing execution time medians of the two series of experiments, the maximum dilation of execution time induced by our profiler was around 4%, but stayed below 1% for the majority of benchmarks.

On Hazel Hen, our production environment, we executed the benchmarks using 16 nodes, with 24 processes on each node. Each benchmark was executed 12 times. Due to the relative small scale of the runs and the sporadic nature of interference, many benchmarks were affected by interference to a smaller degree. Nonetheless, highly interfered runs were encountered and were accurately classified. On the left, Fig. 4b shows the prediction accuracy of benchmark runs, complemented by the percentage-point difference between measured and estimated interference on the right. The figures show that, except for GAPgeofem, the impact of interference was estimated with a high degree of accuracy. GAPgeofem shows again low accuracy, which may again be attributable to its high collective-call frequency. Since the call frequency is measurable, we believe that

it would be generally possible to warn the user of possible inaccuracies in such rare cases. Finding an appropriate threshold, however, is left to future work.

4 Related Work

Performance interference from operating system jitter has been the subject of several studies [2,5,11,17,20]. However, recent work has shown that modern operating systems managed to reduce their noise footprint [16]. Our approach therefore focuses on interferences from other jobs that cause contention on shared resources such as the network or the file system. Moreover, we base our estimates of interference on software and hardware counters that are insensitive to operating system jitter.

At the same time, network and file I/O interference became the focus of more recent studies: Jokanovic et al. attributed loss in network throughput on slim fat trees to inter-application contention [12]. Bhatele et al. observed significant performance variation on Hopper due to neighbor jobs [3]. Yang et al. evaluated different job placement strategies on dragonfly networks to reduce inter-application interference [21]. Similarly, several studies identified variability in applications I/O performance and listed simultaneous file access as one of the possible reasons [14,15,19]. Furthermore, Kuo et. al. investigated how file access patterns influence the degree of I/O contention [13]. All these studies show that simultaneous access to shared resources is a major source of interference, which our method now allows users of HPC systems to quantify.

Mondragon et al. applied extreme value theory to create interference models that predict the execution times of bulk-synchronous applications under interference from OS noise, asynchronous checkpointing, and *in situ* analytics [16]. Our approach estimates the amount of low-frequency but high-impact interference such applications suffer in actual runs with the goal of obtaining performance data with low degrees of interference.

To identify similarity among execution phases of an application for the purpose of performance analysis, Gonzalez et. al. used the density-based clustering algorithm DBScan [8]. To estimate interference impact, we designed a 1D-clustering algorithm based on relative distance.

5 Conclusion

We have demonstrated that we can estimate the impact of interference with high accuracy based on a single run. Our tool chain now provides a warning light to performance analysts that tells them when they need to rerun their experiments because the data they have just collected was subject to interference. It can be integrated with other performance-analysis tools using the P^{n}MPI interface. In the future, we plan to create composite performance profiles free of performance artifacts from multiple interfered measurements. This will allow judging the intrinsic performance of applications in environments where interference is random but due to its frequency unavoidable, making performance measurements (e.g., of different code versions) easier to compare.

Acknowledgment. This work has been supported by the German Research Foundation (DFG) through the Program Performance Engineering for Scientific Software and the ExtraPeak project, by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH16008D, and by the US Department of Energy under Grant No. DE-SC0015524. Additional funding was provided through the Hessian LOEWE initiative within the Software-Factory 4.0 project. Finally, we would like to express our gratitude to Jülich Supercomputing Centre and High Performance Computing Center Stuttgart for giving us access to their supercomputers JUQUEEN and Hazel Hen, respectively.

References

1. Agarwal, S., Garg, R., Vishnoi, N.K.: The impact of noise on the scaling of collectives: a theoretical approach. In: Bader, D.A., Parashar, M., Sridhar, V., Prasanna, V.K. (eds.) HiPC 2005. LNCS, vol. 3769, pp. 280–289. Springer, Heidelberg (2005). https://doi.org/10.1007/11602569_31
2. Beckman, P., Iskra, K., Yoshii, K., Coghlan, S., Nataraj, A.: Benchmarking the effects of operating system interference on extreme-scale parallel machines. Cluster Computing **11**(1), 3–16 (2008)
3. Bhatele, A., Mohror, K., Langer, S.H., Isaacs, K.E.: There goes the neighborhood: performance degradation due to nearby jobs. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC 2013). IEEE Computer Society, November 2013
4. Böhme, D., Geimer, M., Wolf, F., Arnold, L.: Identifying the root causes of wait states in large-scale parallel applications. In: Proceedings of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA, pp. 90–100. IEEE Computer Society, September 2010. https://doi.org/10.1109/ICPP.2010.18
5. De, P., Kothari, R., Mann, V.: Identifying sources of operating system jitter through fine-grained kernel instrumentation. In: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER), pp. 331–340, September 2007
6. Dongarra, J., London, K., Moore, S., Mucci, P., Terpstra, D., You, H., Zhou, M.: Experiences and lessons learned with a portable interface to hardware performance counters. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–6, April 2003
7. Garg, R., De, P.: Impact of noise on scaling of collectives: an empirical evaluation. In: Robert, Y., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2006. LNCS, vol. 4297, pp. 460–471. Springer, Heidelberg (2006). https://doi.org/10.1007/11945918_45
8. Gonzalez, J., Gimenez, J., Labarta, J.: Automatic detection of parallel applications computation phases. In: Proceedings of IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–11, May 2009
9. HLRS: Communication on Cray XC40 Aries network, May 2017. wickie.hlrs.de/platforms/index.php/Communication_on_Cray_XC40_Aries_network
10. Hoefler, T., Schneider, T., Lumsdaine, A.: The impact of network noise at large-scale communication performance. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–8, May 2009

11. Hoefler, T., Schneider, T., Lumsdaine, A.: Characterizing the influence of system noise on large-scale applications by simulation. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC 2010), pp. 1–11. IEEE Computer Society, Washington, DC, USA (2010)
12. Jokanovic, A., Rodriguez, G., Sancho, J.C., Labarta, J.: Impact of inter-application contention in current and future HPC systems. In: Proceedings of the IEEE Symposium on High Performance Interconnects, pp. 15–24, August 2010
13. Kuo, C.S., Shah, A., Nomura, A., Matsouka, S., Wolf, F.: How file access patterns influence interference among cluster applications. In: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER), pp. 1–8 (2014)
14. Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., Allcock, W.: I/O performance challenges at leadership scale. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC 2009), pp. 40:1–40:12. ACM, New York (2009)
15. Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., Wolf, M.: Managing variability in the IO performance of petascale storage systems. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC 2010), pp. 1–12. IEEE Computer Society, Washington, DC, USA (2010)
16. Mondragon, O.H., Bridges, P.G., Levy, S., Ferreira, K.B., Widener, P.: Understanding performance interference in next-generation HPC systems. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC 2016), pp. 384–395, November 2016
17. Petrini, F., Kerbyson, D., Pakin, S.: The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC 2003) (2003)
18. Shah, A., Wolf, F., Zhumatiy, S., Voevodin, V.: Capturing inter-application interference on clusters. In: Proceedings of IEEE International Conference on Cluster Computing (CLUSTER), pp. 1–5, September 2013
19. Shan, H., Shalf, J.: Using IOR to analyze the I/O performance for HPC platforms. In: Cray User Group Conference (2007)
20. Tsafrir, D., Etsion, Y., Feitelson, D.G., Kirkpatrick, S.: System noise, OS clock ticks, and fine-grained parallel applications. In: Proceedings of the 19th annual International Conference on Supercomputing (ICS 2005), pp. 303–312. ACM, New York (2005)
21. Yang, X., Jenkins, J., Mubarak, M., Ross, R.B., Lan, Z.: Watch out for the bully! job interference study on dragonfly network. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC 2016), pp. 750–760, November 2016