A Visualization and Analysis Platform for Performance Tuning

H.M.D. Eranjith, I.D. Fernando, G.K.S. Fernando, W.C.M. Soysa, V.S.D. Jayasena
Department of Computer Science and Engineering
University of Moratuwa
Katubedda, Sri Lanka
{danula.11, isuru.11, kasun.11, madawa.11, sanath}@cse.mrt.ac.lk

***Abstract*—With a framework like OpenTuner, one could build domain-specific multi-objective program auto-tuners and gain significant performance improvements. But explaining why and interpreting the results are often hard, mainly due to the large number of parameters and the inability to figure out how each parameter affects the performance improvement. We have a solution that can explain the performance improvements by identifying key parameters while providing better insights on the tuning process. Our tool uses machine learning techniques to identify parameters which account for a significant performance improvement. A user could utilize different methods provided in the tool to further experiment and verify the accuracy of such findings. Further, our tool uses multidimensional scaling to display all the configurations in a two dimensional graph. This interface allows users to analyze the search space closely and identify clusters of configurations with good or bad performance. It also provides real-time information of tuning process which would help users to optimize the tuning process.**

Keywords— performance tuning; visualization; feature engineering; multidimensional scaling

I. INTRODUCTION

In the present, there are wide range of computer systems that have high processing power with different architectures. It cannot be guaranteed that a program will have consistent performance across multiple computers that have different architectures. Therefore generally a program is tuned on a given system to achieve maximum possible performance on it.

1) Auto Tuners

Changing parameter and flag values and running a program for each is not convenient when there are hundreds of flags and parameters to choose from. As an example, in gcc (GNU C Compiler), there are about 350 flags and parameters in total; in HotSpot JVM [1] there are about 600 flags and parameters. Therefore, it is practically impossible to use manual methods and it requires to automate generating distinct values and run program with each value. Auto-tuners provide a solution for the aforementioned scenario. Auto-tuners achieve the best-performing solution after searching a range of possible configurations and identifying the one that performs better than other configurations.

a) Domain Specific auto-tuners: Most of the auto-tuners are designed for a specific programming language and they cannot be used to tune programs that were implemented using a different programming language. Such auto-tuners are called "domain specific auto-tuners". All the flags, parameters of the programming language (domain) and valid values of each are predefined in domain specific auto tuners. Domain specific auto-tuners are advantageous as user only has to provide the program to be tuned to the auto-tuner.

b) Domain Independent auto-tuners: Domain independent auto-tuners allows users to provide tunable flags and parameters of the domain and valid values of them. Afterwards, user has to define the required application to be tuned and then run the auto-tuner for tuning process.

2) OpenTuner

OpenTuner is an open-source and domain-independent framework that can be used to build domain-specific multi objective auto-tuners. Once the set of flags and parameters are defined with possible values or range of values, OpenTuner generates the configuration space with distinct flag and parameter values using genetic algorithms and inserts into a SQLite database. After that it executes the tuning program for each configuration and records the execution time in SQLite database. On completion of executing program for each configuration, it provides the configuration that had minimum execution time of program as the output [2].

B. Problem

The current implementation of the OpenTuner does not include a user interface for the application users who understand the configuration parameters and configuration space. Also the users do not get enough information to decide whether the currently running configuration will actually optimize the performance of the program. During a search, user cannot see internal information such as how techniques are being used, where in the search space that the current search of OpenTuner is happening, etc.

Above issues will limit the capabilities and also the efficiency as to decide whether a configuration will diverge or converge to an optimized point, a user has to wait until OpenTuner finishes its run and it could sometimes take days to complete. Also sometimes the configuration space of a program will be very large and it is not possible to a user to understand the search space configurations. For example in the Java Virtual Machine there are about 600 flags [3] that can be used to do performance optimization and the search space would be very large. Currently a user cannot get direct information of how the

978-1-5090-0645-8/16/$31.00 ©2016 IEEE

search space is traversed and whether the OpenTuner is moving towards an optimizing direction.

II. Visualization

A. *Visualizing the configuration space*

This section explains the process of visualizing the search space and the results of OpenTuner tuning process, which is one of the major challenges. The traditional auto tuning tools does not provide an insightful interface for users to analyze the configuration space. When we look at the each parameter as a dimension, visualizing configuration space can be viewed as a multidimensional visualization problem. Though there are number of multidimensional visualization techniques available, they are incapable of visualizing a large number of dimensions in a very effective way. Therefore we had to take a different approach other than multidimensional visualization.

Dimensional reduction is one of the key concepts used when it is required to visualize or work with a dataset of large number of dimensions. We successfully visualized the configuration space using four different dimensional reduction techniques.

In our implementation for Isometric feature mapping (Isomap) [3], the reduction happens in three steps. Nearest neighbor search is the initial step where we identify a specific number of nearest neighbors. Second step is shortest-path graph search where we calculate shortest path in the most efficient way using either Dijkstra's Algorithm or the Floyd-Warshall algorithm depending on the data set. Finally, Partial eigenvalue decomposition process reduces the data into a two-dimensional representation.

In the Multidimensional Scaling (MDS) [4] approach, we calculated the dissimilarity between each point using Euclidean distance and then used Scaling by Majorizing a Complicated Function (SMACOF) algorithm to reduce the data set into a two dimensional space.

The method t-distributed Stochastic Neighbor Embedding (t-SNE) [5] which is based on Stochastic Neighbor Embedding [6] converts affinities of data points to probabilities. The affinities in the original space are represented by Gaussian joint probabilities and the affinities in the embedded space are represented by Student's t-distributions.

Locally Linear Embedding (LLE) [7] is another method similar to Isometric feature mapping but focuses on preserving the local structure of data.

B. *Selecting a reduction method*

Fig. 1. shows results obtained after experimenting with the methods described in section A using different datasets. We identified MDS as the most appropriate dimensional reduction method to be used in the visualization tool considering following factors,

1) *Time taken to generate visualization*

As in Table I, Isomap and LLE comparably runs faster than other two methods since they only calculate distances for local neighbors of each point whereas MDS and t-SNE constructs the dissimilarity matrix by calculating distances between all the data points.

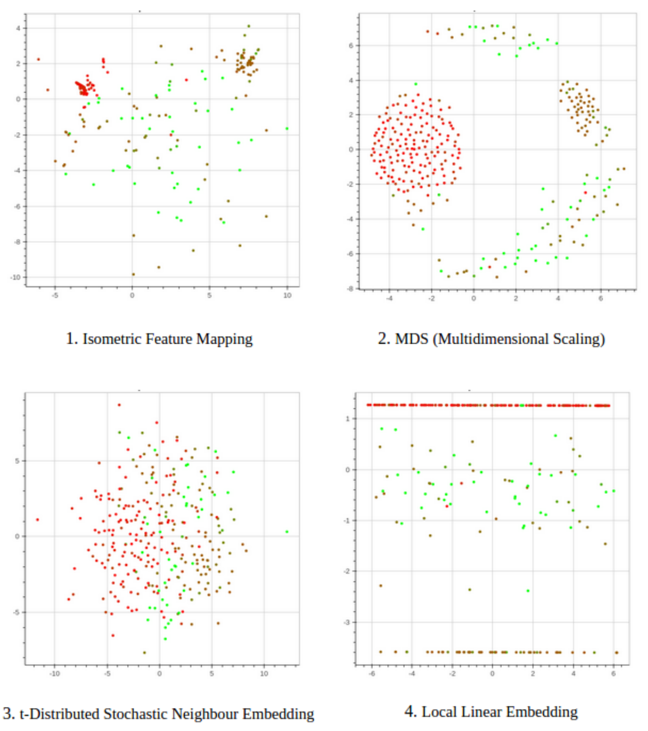

Fig. 1. Results of Multidimensional reduction using different methods

TABLE I. Time (in seconds) to generate visualization

Reduction Method	Number of Configurations		
	300	***500***	***1000***
Isomap	0.110	0.371	1.516
MDS	1.260	6.287	12.998
t-SNE	1.305	4.049	13.437
LLE	0.126	0.307	1.207

2) *Proximity of data points with similar configurations*

The two dimensional representation of the configuration space shown in Fig. 1. needs to preserve the locality of points in the multidimensional space. This helps a user to analyze the impact of certain parameters on performance since two close points represent two configurations which has only few parameters with different values. Generally all four techniques perform well under this criteria.

3) *Tendency to form clusters of good/bad configurations*

For a user to isolate a set of parameters which account for good or bad performance, there should be clusters formed in the visualization. This behavior of forming clusters cannot be enforced during the reduction process but occurs naturally due to different methods used by the particular reduction technique. Also the outcome largely depend on the particular tuning data obtained from OpenTuner.

As seen in Fig. 1. both Isomap and MDS forms clusters with good/bad performance that allow users to effectively identify parameters using the tool. Fig. 2. shows a visualization obtained using MDS for the configurations of tuning gcc parameters for matrix multiplication program.

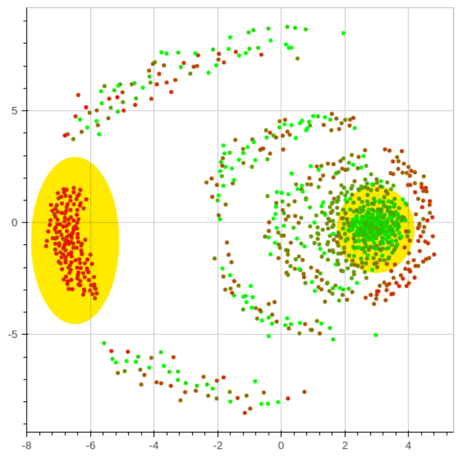

Fig. 2. Formation of clusters in the configuration space.

We could clearly identify a cluster of configurations in red color which has provided good performance and another cluster of configurations in green with bad performance.

III. Analysis

A. Related work

One method of identifying parameters affecting the performance is to see what affect the parameter causes when inverted from the best configuration. For example of HotSpot JVM auto tuning, it is run until completion and the best configuration is looked at [1]. For each JVM performance tuning flag, the best configuration without the performance tuning flag is run to measure the performance. JVM flag that displayed the most performance degradation is considered the flag that affected the performance the most and the flag with the least performance degradation is considered as a flag that affects performance the least.

Advantage of this method is that this method can be used in almost all domain specific tuners because of less number of assumptions on the domain specific tuner. One assumption that is there is that inverting/dropping the parameter is possible. This may not be possible where a parameter cannot be dropped and it has multiple values. Main disadvantage in this method would be that the tuning run has to be run into completion before running this method. With auto tuners having hundreds of parameters, this may take a considerable amount of time to finish. Also this method has to be run on the same host that the tuning run was first run to get correct results.

B. Solution

1) Pre-processing

In order to minimize the time taken for a tuning run, auto tuners extending the functionality of OpenTuner like gcc flags tuner uses a three-valued logic for boolean parameters. For example a gcc flag like "-funroll-loops" can be on or off and specified on the command line or omitted altogether from the command. When the flag is omitted, a default value is chosen based on the other flags. This tri-valued logic is used because in case of gcc flags optimizing, there are conflicting flags and leads to ICEs (Internal Compiler Errors). By adding a default flag, this scenario of conflicting flags is reduced as the default value is chosen internally by gcc to not have a conflict.

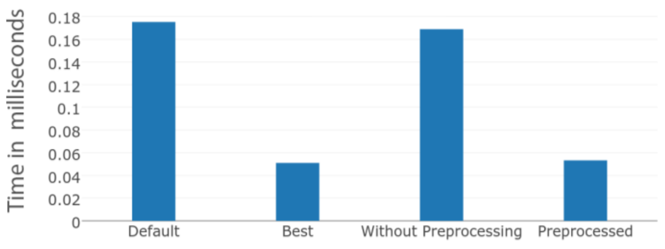

Fig. 3. Timings comparing flag sets obtained by with/without pre-processing.

Problem with this approach in analyzing and visualizing the performance results is that although there are three states, in reality there are only two. To remedy this, before analyzing the results, we have provided a script to modify the database of OpenTuner configurations so that `default` state is replaced with either `on` or `off` by getting the information from gcc via the command line interface `gcc -Q -v <flags> <parameters> source`. This gives the gcc flags enabled by default and by scraping the information given by gcc, all flags with `default` state are replaced by their corresponding `on` and `off` states. Fig. 3. shows the results of using this method. Without using this method the configuration given by machine learning algorithms perform worse than the default configuration while using this method gives a better configuration. Results given in section C uses this method as the pre-processing step.

2) Measuring Importance using statistical correlation

Measuring the correlation between the response and each of the parameter is the simplest method we can use to get an idea of what parameters are significantly correlated with the response. At the initial stage of the project, we used Pearson's correlation and Spearman's Correlation to identify the significance of the parameters. Generally in machine learning domain, features are categorized as follows [8],

a) Relevant: These are features which have an influence on the output and their role can not be assumed by the rest

b) Irrelevant: Irrelevant features are defined as those features not having any influence on the output, and whose values are generated at random for each example.

c) Redundant: A redundancy exists whenever a feature can take the role of another.

One of the major disadvantage in this method is that we cannot identify and eliminate the redundant features using the above method. Even though redundant features can be identified by calculating the whole coefficient matrix, it is infeasible to interpret and eliminate those parameters in a case that has hundreds of parameters.

3) Measuring Importance using Random Forests

Random forest is a popular and very efficient algorithm which belongs to the family of ensemble methods. Random forest is based on model aggregation ideas, for both classification and regression problems [9].

The principle of random forests is to combine many binary decision trees built using several bootstrap samples coming from the training data and choosing randomly at each node a subset of explanatory features. In the random forests framework, the most widely used score of importance of a given variable is the increasing in mean of the error of a tree (Mean Square Error for regression and misclassification rate for classification) in the forest when the observed values of this variable are randomly permuted in the out of bag samples.

In this project we extracted the data from the result database and built a random forest model and calculated the importance score for each parameter. One other advantage in this method is that the operation process of finding the importance score of each parameter does not change with the program or the parameter types. We used the 'caret' package in R and its implementation of Random forest algorithm.

C. Results

For each configuration, timings were taken such that the average has 5% error rate with 95% confidence. Identification of the most important parameters was done using the random forest algorithm. Although the random forest algorithm gives the importance of each flag and parameter, it does not give the value for the flag or parameter that gives the best result. Therefore, to find out the value for the parameter, the best configuration was selected and then the selected flag or parameter values were taken from there.

1) Tuning GNU C Compiler

GCC contains around 360 tunable parameters that would help in improving the running time of a C program. These experiments were done using the GCC auto tuner implemented on OpenTuner by the developers of the OpenTuner. Experiments were carried out on an Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz x86_64 architecture with gcc (Ubuntu 5.2.1-22ubuntu2) 5.2.1 and benchmark program Matrix multiplication. This section presents the results obtained from the experiments carried out.

Matrix multiplication is a benchmark program designed by the developers of OpenTuner. It is a C++ program for multiplying two matrices. Implementation is a serial algorithm with no optimization except transposing the second operand to the multiplication to make good use of the cache.

TABLE II. SIGNIFICANCE OF PARAMETERS GIVEN BY REMOVING PARAMETERS ONE BY ONE FROM THE BEST CONFIGURATION

Parameter	Importance
-finline-small-functions	21.90%
-finline-functions	20.80%
-funsafe-math-optimizations	19.70%
-fwrapv	19.10%
-funroll-loops	2.50%
-foptimize-sibling-calls	2.40%
-ftree-vrp	2.00%
-fbranch-probabilities	1.40%
--param=tracer-min-branch-probability=100	1.40%
--param=max-average-unrolled-insns=20	1.30%
-fno-sched-interblock	0.70%
-fno-tree-forwprop	0.70%

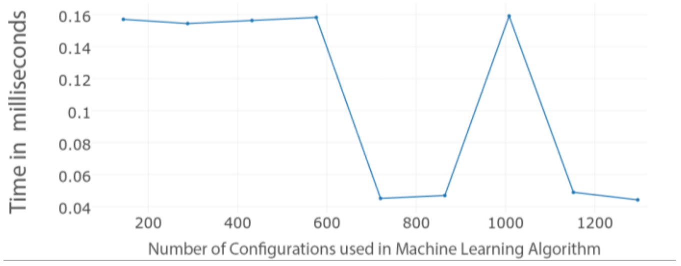

Fig. 4. Running-time of Matrix-multiplication against the size of dataset using randomforest algorithm

OpenTuner developers have given a way to identify the significance of the parameters by recording the running time after removing each flag. Results from this method are shown in Table II. Top 12 most significant parameters were used in compiling the program again and then timings were recorded.

Next, we tried the machine learning algorithms to figure out the significant parameters affecting the performance. With this approach also, we used top 12 most significant parameters. Fig. 4. shows the running times of the benchmark with important parameters found through machine learning algorithm as OpenTuner is testing configurations. Algorithm is run after every 150 configurations that OpenTuner outputs. As more and more configurations are searched by OpenTuner, the results get better, but some outliers are visible. Importance of the parameter was calculated by normalizing the importance value given by the machine learning algorithm.

Top 12 parameters and the corresponding significance given by RandomForest algorithm is given in Table III. Running times of the benchmark is summarized in Fig. 5. Default configuration for GCC with -O3 option takes 0.1531 ms. Best configuration given by OpenTuner takes 0.0428 ms. Reduced set given by the Random Forest algorithm takes 0.0442 ms while removing flags one by one takes 0.0448 ms.

TABLE III. SIGNIFICANCE OF TOP PARAMETERS GIVEN BY RANDOMFOREST ALGORITHM

Parameter	Importance
-ffloat-store	3.32%
-fsignaling-nans	2.08%
-O	1.95%
-ftree-loop-optimize	1.59%
-ftree-loop-vectorize	1.29%
max-once-peeled-insns	1.12%
omega-max-eqs	1.09%
-funsafe-math-optimizations	0.96%
uninit-control-dep-attempts	0.95%
max-modulo-backtrack-attempts	0.86%
selsched-max-lookahead	0.84%
loop-block-tile-size	0.84%

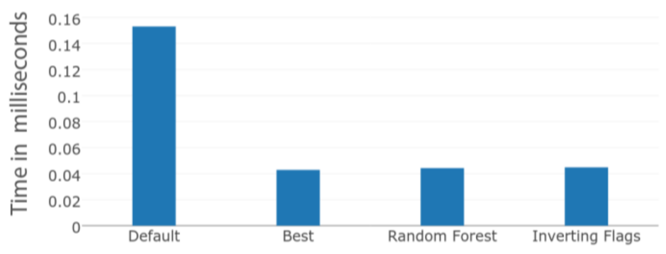

Fig. 5. Running time for the Matrix-multiplication benchmark using several approaches

Therefore the 12 parameters given by the Random Forest algorithm explains 98.73% of the speedup gained by the best configuration which had 360 parameters.

2) Tuning Hotspot JVM

Tuning of the Hotspot JVM [1] was carried out using benchmarks provided by the DaCapo Benchmark Suite. Experiments were carried out on an Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz x86_64 architecture with, OpenJDK java version "1.7.0_91", OpenJDK Runtime Environment (IcedTea 2.6.3) (7u91-2.6.3-0ubuntu0.15.10.1) and OpenJDK 64-Bit Server VM (build 24.91-b01, mixed mode)

DaCapo 9.12 lusearch benchmark [10] is a benchmark which uses Apache Lucene to run 64 text search queries in the works of Shakespeare and the King James Bible. OpenTuner was run for 3 hours to get the best configuration and there was a speedup of 47.36 % compared to the default value. In this section we will look at how to figure out what caused this speedup.

First, flag importance was calculated using the OpenTuner method of removing flags one by one from the best configuration. Running time of the configuration without the flag was measured and then importance was calculated as in (1) and (2).

TABLE IV. SIGNIFICANCE OF PARAMETERS GIVEN BY REMOVING PARAMETERS ONE BY ONE FROM THE BEST CONFIGURATION

Parameter	Importance
-XX:-NeedsDeoptSuspend	1.99%
-XX:-UseGCTaskAffinity	1.93%
-XX:-DumpSharedSpaces	1.87%
-XX:+UseSharedSpaces	1.76%
-XX:GCTaskTimeStampEntries=20	1.63%
-XX:-DisableExplicitGC	1.59%
-XX:-ParallelRefProcEnabled	1.42%
-XX:InteriorEntryAlignment=17	1.40%
-XX:-ReduceInitialCardMarks	1.34%
-XX:-RequireSharedSpaces	1.33%
-XX:-NeedsDeoptSuspend	1.99%
-XX:-UseGCTaskAffinity	1.93%

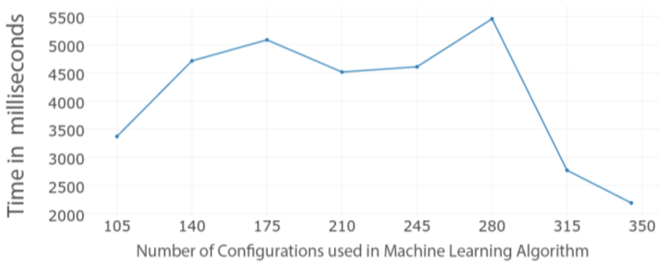

Fig. 6. Running-time of DaCapo 9.12 lusearch against the size of dataset using randomforest algorithm

Impact = Running Time of Configuration – Running Time of Best Configuration (1)

Importance = Impact * 100 / Total Impact (2)

When the benchmark was run with the above 10 parameters, the JVM crashed and the parameter -XX:InteriorEntryAlignment=17 had to be removed to make the JVM run normally into completion.

RandomForest algorithm was also run to get the importance of the parameters for the benchmark and the algorithm was run after every 35 configurations in the tuning run and DaCapo lusearch benchmark running time was measured with the most important 10 parameters that the algorithm output as shown in Fig. 6.

Most important parameters that the random forest algorithm gave is listed in Table V.

TABLE V. SIGNIFICANCE OF TOP PARAMETERS GIVEN BY RANDOMFOREST ALGORITHM

Parameter	Importance
-XX:StackShadowPages=20	0.90%
-XX:OldPLABWeight=25	0.61%
-XX:NodeLimitFudgeFactor=1493	0.56%
-XX:CodeCacheExpansionSize=33889	0.53%
-XX:-ProfileInterpreter	0.49%
-XX:ParallelOldDeadWoodLimiterStdDev=44	0.47%
-XX:LargePageHeapSizeThreshold=73320140	0.46%
-XX:MaxRAM=178918540219	0.43%
-XX:-UseInlineCaches	0.41%

TABLE VI. SIGNIFICANT PARAMETERS WITH DEFAULT VALUES

Parameter	Importance
-XX:+UseCompiler	1.13%
-XX:Tier3CompileThreshold=2000	0.73%
-XX:Tier4InvocationThreshold=5000	0.62%
-XX:G1UpdateBufferSize=256	0.52%
-XX:Tier4MinInvocationThreshold=600	0.52%

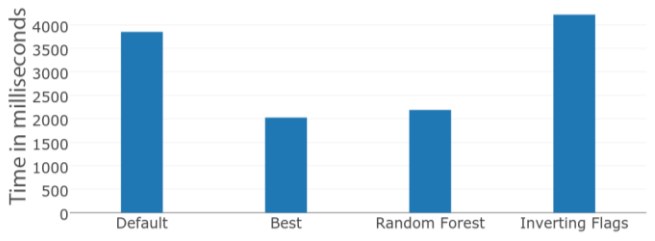

Fig. 7. Running time for the DaCapo 9.12 lusearch benchmark using several approaches

Parameters in Table VI were also given as important to the running-time, but they have values equal to the default and hence it can be concluded that these parameters affect the performance in a negative way when the value is not the default for the JVM.

Running time of the benchmark is summarized in Fig. 7. Default configuration for JVM without any tuning options takes 3853.2 ms. Best configuration given by OpenTuner takes 2028.3 ms. Reduced set given by the Random Forest algorithm takes 2189.8 ms while removing flags one by one takes 4222.2 ms. Therefore the 9 parameters given by the Random Forest algorithm explains 91.15% of the speedup gained by the best configuration which had 412 parameters.

IV. Discussion And Conclusion

We set out to develop a minimal invasive visualization and analysis tool for OpenTuner. This tool had to be generic enough for handle all kinds of domain specific tuners that extend the functionality of OpenTuner. Also the tool could not have any impact on the tuning process, but allow the user to visualize and analyze while the tuning process is going on.

One major drawback of existing methods in OpenTuner for analysis was that it could be done only after the tuning run is over in the same workstation that the tuning was going on because the tuning program had to be run again for the analysis. Analysis and visualization tool presented can be used remotely in another workstation without interrupting the tuning process as it uses only the data that is already there and uses machine learning techniques for analysis.

Machine learning techniques are highly sensitive to data and even though there is a very good speedup in tuning, it takes lots more configurations for a good configuration to come out of the machine learning techniques. Reason for this is that while OpenTuner is searching the configuration space, OpenTuner might happen upon a configuration by chance and that configuration might be the best configuration. For the machine learning technique to identify the best parameters, OpenTuner has to search around this best configuration as well. As searches around the best configurations are fed into the machine learning algorithm, the algorithm can come into conclusions about the parameters with more accuracy.

Visualization is also very important to get an idea about the tuning process. There's no visualization in OpenTuner and we introduced two interfaces for OpenTuner where a user could get insights on the tuning process. Our primary interface, which consists of a graph with execution time against the runtime of OpenTuner is aimed at users who want information on the progress of tuning process. The second interface, obtained by applying dimensional reduction to the data set is more relevant for advanced users who wish to analyse the tuning process by examining the configuration space.

The two major objectives of the project was to develop an analysis platform and a visualization tool for OpenTuner and to explain the importance of parameters towards the performance improvement. Our solution contains two novel approaches in achieving these two objectives. We used Multi Dimensional Scaling as a dimensional reduction technique to visualize the configuration space in a two dimensional graph. We successfully applied random forest as a machine learning techniques to identify parameters which have a significant impact towards performance.

Our tool provide ample support for a user to understand the tuning process and obtain various results and optimizations through that. Key features provided in the tool are visualizing the configuration space and tuning process in real time, comparing individual configurations or regions, identifying important parameters using machine learning techniques and analyzing the effect on performance for a user specified configuration.

References

[1] S. Jayasena, M. Fernando, T. Rusira, C. Perera, and C. Philips, "Auto-tuning the Java Virtual Machine," in Proceedings of the IEEE International Workshop on Automatic Performance Tuning 2015 (iWAPT2015), 2015, pp. 1261–1270.

[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in Proceedings of the 23rd international conference on Parallel architectures and compilation, 2014, pp. 303–316.

[3] J. B. Tenenbaum, V. De Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," Science (80-.)., vol. 290, no. 5500, pp. 2319–2323, 2000.

[4] J. B. Kruskal, "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis," Psychometrika, vol. 29, no. 1, pp. 1–27, 1964.

[5] L. J. P. Van Der Maaten and G. E. Hinton, "Visualizing high-dimensional data using t-sne," J. Mach. Learn. Res., vol. 9, no. 85, pp. 2579–2605, 2008.

[6] G. E. Hinton and S. T. Roweis, "Stochastic neighbor embedding," in Advances in Neural Information Processing Systems 15, 2002, pp. 833–840.

[7] S. T. Roweis and L. K. Saul, "Nonlinear Dimensionality Reduction by Locally Linear Embedding," Sci. New Ser., vol. 290, no. 5500, pp. 2323–2326, 2000.

[8] M. Hall, "Correlation-based feature selection for machine learning," The University of Waikato, 1999.

[9] J.-M. P. Robin Genuer and C. Tuleau-Malot, "Variable Selection using Random Forests," Pattern Recognitions Lett. 31, vol. 31, no. 14, pp. 2225–2236, Nov. 2010.

[10] S. M. Blackburn, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, R. Garner, D. von Dincklage, B. Wiedermann, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, and D. Frampton, "The DaCapo benchmarks," ACM SIGPLAN Not., vol. 41, no. 10, p. 169, 2006.