A Design of Experiments Approach to Autotuning under Tight Budget Constraints

Pedro Bruel*†, Arnaud Legrand*, Brice Videau* and Alfredo Goldman†
*University of Grenoble Alpes, CNRS, INRIA, LIG - Grenoble, France
Email: {arnaud.legrand, brice.videau}@imag.fr
†University of São Paulo - São Paulo, Brazil
Email: {phrb, gold}@ime.usp.br

***Abstract*—Abstract**

I. Introduction

Optimizing code for objectives such as performance and power consumption is fundamental to the success and cost effectiveness of industrial and scientific endeavors in High Performance Computing. A considerable amount of highly specialized time and effort is spent in porting and optimizing code for GPUs, FPGAs and other hardware accelerators. Experts are also needed to leverage bleeding edge software improvements in compilers, languages, libraries and frameworks. The automatic configuration and optimization of High Performance Computing applications, or *autotuning*, is a technique effective in decreasing the cost and time needed to adopt efficient hardware and software. Typical targets for autotuning include algorithm selection, source-to-source transformations and compiler configuration.

Autotuning can be studied as a search problem, where the objective is to minimize single or multiple software of hardware metrics. The exploration of the search spaces defined by configurations and optimizations present interesting challenges to search strategies. These search spaces grow exponentially with the number of considered configuration parameters and their possible values. They are also difficult to extensively explore due to the often prohibitive costs of hardware utilization and program compilation and execution times. Developing autotuning strategies capable of producing good optimizations while minimizing resource utilization is therefore essential. The capability of acquiring knowledge about an optimization problem is also a desired feature of an autotuning strategy, since this knowledge can decrease the cost of subsequent optimizations of the same application or for the same hardware.

It is common and usually effective to use search meta-heuristics such as genetic algorithms and simulated annealing in autotuning. These strategies usually attempt to exploit local properties and are not capable of fully exploiting global search space structures. They are also not much more effective in comparison with a naive uniform random sample of the search space [1], [2], and usually rely on a large number of measurements and frequent restarts to achieve good performance improvements. Search strategies based on gradient descent also are commonly used in autotuning and rely on a large number of measurements. Their effectiveness diminishes additionally in search spaces with complex local structures. Completely automated machine learning autotuning strategies are effective in building models for predicting important optimization parameters, but still rely on a sizable data set for training. Large data sets are fundamental to strategies based on machine learning since they select models from a generally very large class.

Search strategies based on meta-heuristics, gradient descent and machine learning require a large number of measurements to be effective, and are usually incapable of providing knowledge about search spaces to users. At the end of each autotuning session it is difficult to decide if and where further exploration is warranted, and impossible to know which parameters are responsible for the observed improvements. After exploring a search space, it is impossible to confidently deduce its global properties since its was explored with unknown biases.

In this paper we propose an autotuning strategy that leverages existing expert and approximate knowledge about a problem in the form of a performance model, and refines this initial model iteratively using empirical performance evaluations, statistical analysis and user input. Our strategy puts a heavy weight on decreasing the costs of autotuning by using efficient Design of Experiments strategies to minimize the number of experiments needed to find good optimizations. Each optimization iteration uses *Analysis of Variance* (ANOVA) to help identify the relative significance of each configurable parameter to the performance observations. An architecture- and problem-specific performance model is built iteratively and with user input, enabling informed decisions on which regions of the search space are worth exploring.

We present the performance of our approach on a Laplacian Kernel for GPUs where the search space, global optimum and performance model approximation are known. The experimental budget on this application were tightly constrained. The speedups achieved and the budget utilization of our approach on this setting motivated a more comprehensive performance evaluation. We chose the *Search Problems in Automatic Performance Tuning* (SPAPT) [3] benchmark for this evaluation, where our approach was able to find speedups of over $50\times$ for some SPAPT applications, finding speedups better than random sampling in some scenarios. Despite using

generic performance models for every SPAPT application, our approach was able to significantly decrease the budget used to find performance improvements.

The rest of this paper is organized as follows. Section II presents related work on source-to-source transformation, which is the main optimization target in SPAPT problems, on autotuning systems and on search space exploration strategies. Section V presents a detailed description of the implementation of our approach and its background. It discusses the Design of Experiments concepts we incorporate, and the ANOVA and linear regression algorithms we use in analysis steps. Section VI presents our results with the GPU Laplacian Kernel and the SPAPT benchmark. Section VII discusses our conclusions and future work.

II. Background

A. Source-to-source Transformation

B. Autotuning

John Rice's Algorithm Selection framework [4] is the precursor of autotuners in various problem domains. In 1997, the PHiPAC system [5] used code generators and search scripts to automatically generate high performance code for matrix multiplication. Since then, systems approached different domains with a variety of strategies. Dongarra *et al.* [6] introduced the ATLAS project, that optimizes dense matrix multiplication routines. The OSKI [7] library provides automatically tuned kernels for sparse matrices. The FFTW [8] library provides tuned C subroutines for computing the Discrete Fourier Transform. Periscope [9] is a distributed online autotuner for parallel systems and single-node performance. In an effort to provide a common representation of multiple parallel programming models, the INSIEME compiler project [10] implements abstractions for OpenMP, MPI and OpenCL, and generates optimized parallel code for heterogeneous multi-core architectures.

A different approach is to combine generic search algorithms and problem representation data structures in a single system that enables the implementation of autotuners for different domains. The PetaBricks [11] project provides a language, compiler and autotuner, enabling the definition and selection of multiple algorithms for the same problem. The ParamILS framework [12] applies stochastic local search algorithms to algorithm configuration and parameter tuning. The OpenTuner framework [13] provides ensembles of techniques that search the same space in parallel, while exploration is managed by an implementation of a solver of the multi-armed bandit problem.

C. Search Space Exploration Strategies

III. Design of Experiments

An *experimental design* determines a selection of experiments whose objective is to identify the relationships between *factors* and *responses*. While factors and responses can refer to different concrete entities in other domains, in computer experiments factors can be configuration parameters for algorithms and compilers, for example, and responses can be the execution time or memory consumption of a program. Each possible value of a factor is called a *level*.

Experimental designs are with objectives such as identifying the most important factors and building an analytical model for the response. The remaining of this section discusses some design construction techniques we explored and presents the technique we selected for our approach.

A. Design Construction Techniques

The application of Design of Experiments to autotuning problems requires design construction techniques that support factors of different types and number of possible values. Autotuning problems typically combine factors such as binary flags, integer and floating point numerical values, and unordered enumerations of abstract values. Minimizing the number of experiments needed to find good optimizations is a also a fundamental requirement since we are interested in autotuning for scenarios with tight budget constraints.

The design construction techniques that fit these requirements are limited. Designs that simply test all possible factor combinations, or *full factorial designs*, would provide complete information about the global minimum but are unfeasible for most autotuning problems. In the *2-level screening with random level sampling* technique, factors with more than two unordered levels are sampled at two random levels. This enables using small design such as the Plackett-Burman [14] screening design. Advantages are the small design size and good estimation capability for main effects. Incapability of estimating interactions is a disadvantage of this strategy, but the main drawback is the lack of information for levels not selected in the initial screening.

In *contractive replacement*, an initial 2-level design is used to generate mixed-level designs by re-encoding columns into a new single column representing a multi-level factor. The contractive replacement of Addelman-Kempthorne [15] is a strategy of this kind. Advantages of this technique are the small design sizes and the ability to estimate main effects. Additionally, the contractive replacement technique preserves orthogonality. Due to strict requirements on initial designs, not all 2-level designs can be contracted.

The *direct generation* algorithm presented by Grömping and Fontana [16] enables the generation of multi-level designs by solving Mixed Integer Problems (MIP). The advantages of this technique are the direct generation of multi-level designs and a clearly defined optimality criterion. Since this construction relies on solving carefully formulated MIP problems, it presents strong restrictions on the size and shape of the designs that can be generated.

IV. D-Optimal Designs

D-Optimal designs are the class of designs that best fits our requirements of supporting multi-level factors and minimizing the number of experiments. The algorithms for constructing D-Optimal designs are relatively fast and have few restrictions.

```
rosenbrock <- function(x, y) {
    return(((1.0 - x) ^ 2) + (100.0 * ((y - (x ^ 2)) ^ 2)))
}
```

Figure 1: Defining the Rosenbrock function in R

It is necessary to select a model that relates factors and responses to construct a D-Optimal design. The model selection can be based on previous experiments or on expert knowledge of the problem. Once a model is selected, algorithmic construction is performed by searching for the set of experiments that minimizes the *D-Optimality* criterion, a measure of the *variance* of the *estimators* of the *regression coefficients* associated with the selected model. This search is usually done by swapping experiments from the current candidate set with experiments from a pool of possible experiments, according to certain rules, until some stopping criterion is met. In the approach presented in this paper we used Fedorov's algorithm [17] for constructing D-Optimal designs, implemented in R in the AlgDesign package.

Considering that we are going to analyze the results of an experiments plan, the *D-Efficiency* of a design is inversely proportional to the *geometric mean* of the *eigenvalues* of the plan's *covariance matrix*. A D-Optimal design has the best D-Efficiency. Our current approach is based on D-Optimal designs.

Table I: Comparison of 3 optimization methods on Rosenbrock's function, using a buget of 10 points with 100 repetitions

Method	Mean	Min.	Max.
Random Sampling	1.3×10^2	1.2	1.5×10^3
D-Opt. w/ Linear Model	1.4×10^4	1.4×10^4	1.4×10^4
D-Opt. w/ Correct Model	0	0	0

Table II: Shortened ANOVA table for the fit of the correct model using 10 experiments

	F value	Pr(>F)
x	9.4×10^{28}	7.6×10^{-44}
y	1×10^{32}	2.1×10^{-48}
I(x^4)	2.3×10^{32}	6.4×10^{-49}
I(x^2)	2.4×10^{30}	5.8×10^{-46}
I(y^2)	6.8×10^{29}	3.9×10^{-45}
y:I(x^2)	4.5×10^{31}	7.3×10^{-48}

Table III: Shortened ANOVA table for the fit of the naive linear model using 10 experiments

	F value	Pr(>F)
x	7.5×10^{-6}	1
y	1.4	0.27

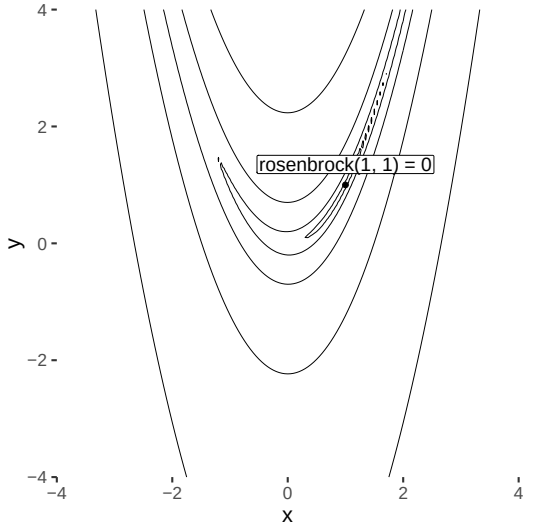
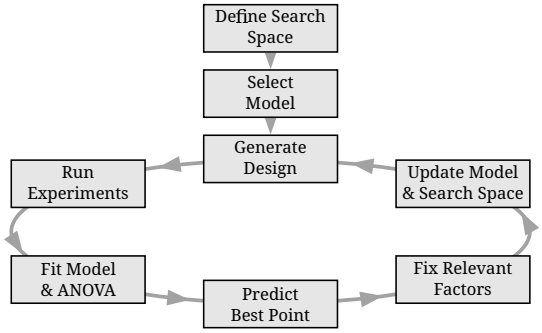


Figure 2: Contour plot in log_{10} scale and global optimum of Rosenbrock's function

V. Applying Design of Experiments to Autotuning

A. ANOVA

B. The DLMT Strategy



VI. Performance Evaluations

A. Example on a GPU Laplacian Kernel

Table IV: Comparison of 7 optimization methods on the Laplacian Kernel, using a budget of 125 points with 1000 repetitions

	Mean	Min.	Max.	Mean Points	Max Points
RS	1.10	1.00	1.39	120.00	120.00
LHS	1.17	1.00	1.52	98.92	125.00
GS	6.46	1.00	124.76	22.17	106.00
GSR	1.23	1.00	3.16	120.00	120.00
GA	1.12	1.00	1.65	120.00	120.00
LM	1.02	1.01	3.77	119.00	119.00
DLMT	1.01	1.01	1.01	54.84	56.00

1) Results:

B. Results on the SPAPT Benchmark

1) The SPAPT Benchmark:

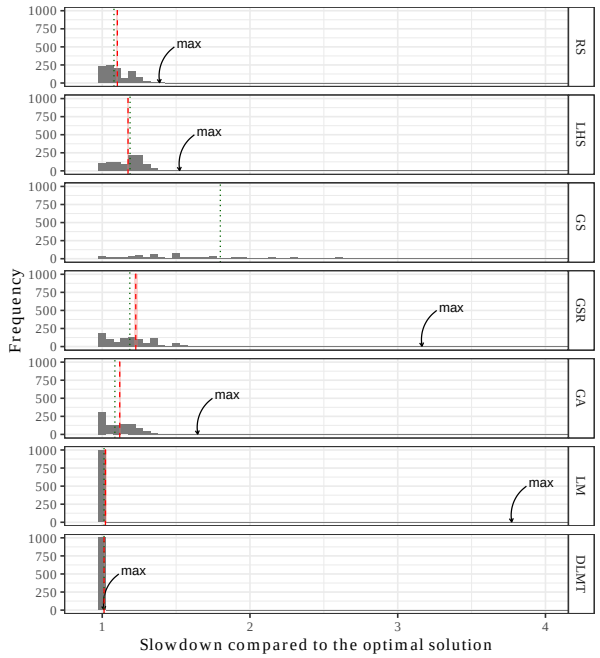


Figure 3: Results

Table V: Set of applications we used from the SPAPT benchmark

Kernel	Operation	Factors	Size
`atax`	Matrix transp. & vector mult.	18	2.6×10^{16}
`dgemv3`	Scalar, vector & matrix mult.	49	3.8×10^{36}
`gemver`	Vector mult. & matrix add.	24	2.6×10^{22}
`gesummv`	Scalar, vector, & matrix mult.	11	5.3×10^{9}
`hessian`	Hessian computation	9	3.7×10^{7}
`mm`	Matrix multiplication	13	1.2×10^{12}
`mvt`	Matrix vector product & transp.	12	1.1×10^{9}
`tensor`	Tensor matrix mult.	20	1.2×10^{19}
`trmm`	Triangular matrix operations	25	3.7×10^{23}
`bicg`	Subkernel of BiCGStab	13	3.2×10^{11}
`lu`	LU decomposition	14	9.6×10^{12}
`adi`	Matrix sub., mult., & div.	20	6.0×10^{15}
`jacobi`	1-D Jacobi computation	11	5.3×10^{9}
`seidel`	Matrix factorization	15	1.3×10^{14}
`stencil3d`	3-D stencil computation	29	9.7×10^{27}
`correlation`	Correlation computation	21	4.5×10^{17}

2) Experimental Methodology:

3) Results:

VII. Conclusion

Acknowledgment

References

[1] K. Seymour, H. You, and J. Dongarra, "A comparison of search heuristics for empirical code optimization." in *CLUSTER*, 2008, pp. 421–429.

[2] P. M. Knijnenburg, T. Kisuki, and M. F. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," *The Journal of Supercomputing*, vol. 24, no. 1, pp. 43–67, 2003.

[3] P. Balaprakash, S. M. Wild, and B. Norris, "Spapt: Search problems in automatic performance tuning," *Procedia Computer Science*, vol. 9, pp. 1959–1968, 2012.

[4] J. R. Rice, "The algorithm selection problem," in *Advances in Computers 15*, 1976, pp. 65–118.

[5] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology," in *Proceedings of International Conference on Supercomputing, Vienna, Austria*, 1997.

[6] J. J. Dongarra and C. R. Whaley, "Automatically tuned linear algebra software (atlas)," *Proceedings of SC*, vol. 98, 1998.

[7] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[8] M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.

[9] M. Gerndt and M. Ott, "Automatic performance analysis with periscope," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 736–748, 2010.

[10] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–12.

[11] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, *PetaBricks: a language and compiler for algorithmic choice*. ACM, 2009, vol. 44, no. 6.

[12] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: an automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, 2009.

[13] 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*. ACM, 2014, pp. 303–316.

[14] R. L. Plackett and J. P. Burman, "The design of optimum multifactorial experiments," *Biometrika*, vol. 33, no. 4, pp. 305–325, 1946.

[15] S. Addelman and O. Kempthorne, "Some main-effect plans and orthogonal arrays of strength two," *The Annals of Mathematical Statistics*, pp. 1167–1176, 1961.

[16] U. Grömping and R. Fontana, "An algorithm for generating good mixed level factorial designs," Beuth University of Applied Sciences, Berlin, Tech. Rep., 2018.

[17] V. V. Fedorov, *Theory of optimal experiments*. Elsevier, 1972.

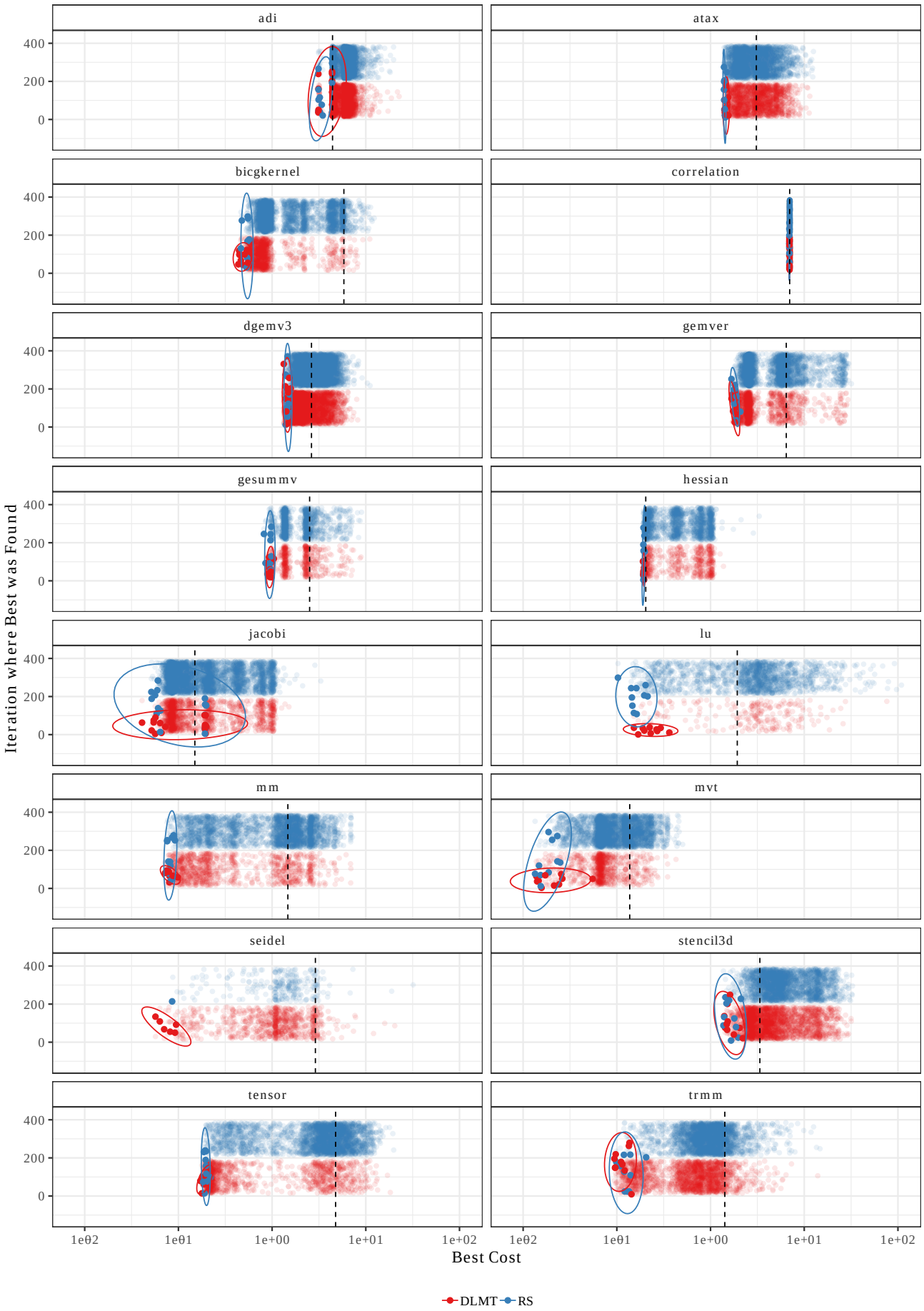


Figure 4: Results