Methods for Efficient Resource Utilization in Statistical Machine Learning Algorithms

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Helena Victoria Kotthaus

Dortmund

2018

Tag der mündlichen Prüfung:	7. Juni 2018
Dekan / Dekanin:	Prof. Dr. Gernot Fink
Gutachter / Gutachterinnen:	Prof. Dr. Peter Marwedel
	Prof. Dr. Jörg Rahnenführer

Acknowledgments

First and foremost, I would like to thank my advisor Prof. Peter Marwedel for his support, motivation and guidance over the past seven years. Thank you for pointing my research to a rewarding direction and providing me with the opportunity to work on this beautiful interdisciplinary field of research. I would also like to thank Prof. Jörg Rahnenführer, for his commitment to participate as a second reviewer of my thesis.

Special thanks go out to Prof. Jan Vitek for many fruitful and stimulating discussions about speeding up statistical algorithms and to Prof. Floréal Morandat for giving me the opportunity to work on the traceR profiling tool.

I would furthermore like to thank Prof. Uwe Ligges for introducing me to the members of the Core R Team, which enabled me to gain detailed insights of the development process of the R language.

My PhD studies would have been less exciting without the summer internship at Oracle Labs, which had a great impact on my further development. I owe special thanks to Prof. Pinar Tözün and to Prof. Michael Engel – thank you for this unforgettable experience.

The development of the resource-aware model-based optimization framework presented in this thesis would not have been possible without the previous work of numerous colleagues at the statistics department of TU Dortmund University. In this context, I owe special thanks to Jakob Richter, Michel Lang and Prof. Bernd Bischl. On the implementation side, my work was further supported by Ingo Korb, Andreas Lang, Felix Gonsior and Markus Künne.

The research leading to this thesis has received funding from the Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 - Providing Information by Resource-Constrained Data Analysis - project A3.

I would furthermore like to thank all my colleagues, in particular those from Dortmund, for many technical and fun conversations that brightened up my day-to-day life. Special thanks go out to Malte Isberner, thank you for the time where you offered me your beautiful working desk at your home in Silicon Valley for starting the writing process of my thesis as well as for your encouragement and feedback. I would also like to thank Christian Bockerman, thank you for tirelessly supporting me during the writing process of my thesis.

Many thanks go to Prof. Helmut Dohmann who pushed me to move to TU Dortmund University for my Master studies, and to my sister Catherine Kotthaus for motivating me to pick up computer science as a profession.

Last but not least, I want to thank my family, my friends and my boyfriend for their unconditional support and patience during the course of this thesis.

Abstract

In recent years, statistical machine learning has emerged as a key technique for tackling problems that elude a classic algorithmic approach. One such problem, with a major impact on human life, is the analysis of complex biomedical data. Solving this problem in a fast and efficient manner is of major importance, as it enables, e.g., the prediction of the efficacy of different drugs for therapy selection. While achieving the highest possible prediction quality appears desirable, doing so is often simply infeasible due to resource constraints. Statistical learning algorithms for predicting the health status of a patient or for finding the best algorithm configuration for the prediction require an excessively high amount of resources. Furthermore, these algorithms are often implemented with no awareness of the underlying system architecture, which leads to sub-optimal resource utilization.

This thesis presents methods for efficient resource utilization of statistical learning applications. The goal is to reduce the resource demands of these algorithms to meet a given time budget while simultaneously preserving the prediction quality. As a first step, the resource consumption characteristics of learning algorithms are analyzed, as well as their scheduling on underlying parallel architectures, in order to develop optimizations that enable these algorithms to scale to larger problem sizes. For this purpose, new profiling mechanisms are incorporated into a holistic profiling framework. The results show that one major contributor to the resource issues is memory consumption. To overcome this obstacle, a new optimization based on dynamic sharing of memory is developed that speeds up computation by several orders of magnitude in situations when available main memory is the bottleneck, leading to swapping out memory.

One important application that can be applied for automated parameter tuning of learning algorithms is model-based optimization. Within a huge search space, algorithm configurations are evaluated to find the configuration with the best prediction quality. An important step towards better managing this search space is to parallelize the search process itself. However, a high runtime variance within the configuration space can cause inefficient resource utilization. For this purpose, new resource-aware scheduling strategies are developed that efficiently map evaluations of configurations to the parallel architecture, depending on their resource demands. In contrast to classical scheduling problems, the new scheduling interacts with the configuration proposal mechanism to select configurations with suitable resource demands. With these strategies, it becomes possible to make use of the full potential of parallel architectures. Compared to established parallel execution models, the results show that the new approach enables model-based optimization to converge faster to the optimum within a given time budget.

Contents

1 Introduction **1**
- 1.1 Challenges in Statistical Learning Applications 1
- 1.2 Selected Contributions to Statistical Learning 2
  - 1.2.1 The Domain Specific Language R 2
  - 1.2.2 Model-Based Optimization 4
- 1.3 Contribution of this Thesis 5
- 1.4 Outline of this Thesis 6
- 1.5 Author's Contribution of this Thesis 8
- 1.6 Publications Covered by this Thesis 9

2 Fundamentals: Model-Based Optimization and the R Language **13**
- 2.1 Model-Based Optimization 14
  - 2.1.1 Hyperparameter Optimization 15
  - 2.1.2 Model-Based Optimization Algorithm 16
- 2.2 The R Language Environment 19
  - 2.2.1 Language Characteristics 21
  - 2.2.2 Execution Model 22
- 2.3 Summary 23

3 Profiling of Machine Learning Algorithms **25**
- 3.1 Optimizations for R - Existing Approaches 26
- 3.2 Profiling Tool for R - traceR 28
- 3.3 Profiling of Machine Learning Algorithms Written in R 29
  - 3.3.1 Experimental Setup 29
  - 3.3.2 Runtime Behavior Analysis 32
  - 3.3.3 Memory Consumption Analysis 38
- 3.4 Summary 43

4 Efficient Memory Utilization for Machine Learning Algorithms **45**
- 4.1 R Optimizations and Memory Footprint Reduction - Existing Approaches 46
- 4.2 Memory Allocation in the R Runtime Environment - Fundamentals 48
- 4.3 Memory Footprint Reduction via Dynamic Page Sharing Strategies 50
  - 4.3.1 Page Duplication Avoidance 50
  - 4.3.2 Static Refinement via Annotations 53
  - 4.3.3 Dynamic Refinement via Page Contents 54
  - 4.3.4 Dynamic Page Sharing Optimization 55
  - 4.3.5 Evaluation 59
- 4.4 Summary 71

5 Profiling of Parallel Machine Learning Algorithms **73**
5.1 Parallel Execution Mechanisms in R
- Fundamentals 74
5.2 Parallel Profiling Mechanisms - traceR 75
5.2.1 Single Machine Profile 76
5.2.2 Multiple Machine Profile 77
5.3 Profiling of Parallel Machine Learning Algorithms 79
5.3.1 Experimental Setup 79
5.3.2 Runtime Behavior Analysis 79
5.3.3 Memory Consumption and CPU Utilization Analysis . . . 82
5.4 Summary 83

6 Resource-Aware Scheduling Strategies for Parallel Machine Learning Algorithms **85**
6.1 Parallel Execution of Model Based Optimization
- Existing Approaches 87
6.1.1 Parallel Synchronous Execution 88
6.1.2 Parallel Asynchronous Execution 90
6.2 Resource-Aware Model-Based Optimization 92
6.3 Resource-Aware Scheduling Strategies 94
6.3.1 First Fit Scheduling Strategy 95
6.3.2 Knapsack based Scheduling Strategy 99
6.3.3 Refinement of Job Priorities 100
6.3.4 Evaluation 101
6.4 Resource-Aware Scheduling Strategies for Heterogeneous Embedded Architectures 112
6.4.1 Knapsack based Scheduling Strategy for Heterogeneous Architectures 113
6.4.2 Evaluation 115
6.5 Summary 119

7 Conclusion and Outlook **121**
7.1 Summary of Research Contributions 121
7.2 Future Research 124

Bibliography **127**

List of Figures **142**

List of Tables **144**

Chapter 1

Introduction

1.1 Challenges in Statistical Learning Applications

In recent years, *statistical machine learning* has emerged as a key technique for tackling problems that elude a classic algorithmic approach. One such problem, with a major impact on human life, is the analysis of complex biomedical data. Here, the complexity and the dimensionality of the data that has to be analyzed is increasing due to new high-throughput technologies such as micro-arrays. Most datasets are high-dimensional, consisting of a large number of features. For example, when performing statistical analysis of medical records, the number of monitored gene expressions for each patient is so large that the resulting data set is huge even if the number of patients is fairly small [BG11].

Finding a model that describes the interaction between the high-dimensional variables to identify important features and their interplay is a complex problem. Solving this problem in a fast and efficient manner is of major importance, as it enables, e.g., the prediction of the influence of different drugs for therapy selection or the prediction of the survival time of a patient under a given medication. While achieving the highest possible prediction quality appears desirable, it is often simply infeasible due to *time and budget constraints*. This applies especially in the field of personalized medicine, where the patient needs a treatment as soon as possible. Typically, the trade-off between prediction quality and resource utilization can be tuned by changing parameters of these algorithms. However, the correlation between parameter values and both prediction quality and resource utilization is often highly non-linear, such that finding a suitable parameter configuration is a challenging problem in itself.

Statistical learning algorithms like classification for predicting the health status of a patient or model-based optimization for finding the best algorithm configuration for the prediction require an excessively high amount of resources. For the optimization of such algorithms, the overall goal is to reduce resource demands to meet a given time budget while simultaneously preserving the prediction quality. There exist several different approaches that try to accomplish this goal.

Many existing approaches that tackle the problem of learning under resource constraints seek to optimize the algorithm itself, thereby reducing its resource demands. This naturally requires a thorough understanding of the theoretical underpinnings of the algorithm that range far beyond the mere application [TS13; NS17]. Another approach is to view the algorithm as a black-box and optimize the

resource utilization by enhancing the runtime and memory consumption behavior of the software environment, and the language the algorithm was implemented in. In contrast to the first approach, the second approach – on which we focus in this thesis – can be applied to a broad range of learning algorithms without the need of detailed knowledge of the algorithm itself. However, it naturally only yields benefits for algorithms that are implemented in the chosen language or rely on the chosen runtime.

1.2 Selected Contributions to Statistical Learning

The most widely used programming language for statistical data analysis - and biostatistics in particular - is the *GNU R language* [IG96]. While apparently not affecting its popularity, its lavish use of resources makes it unsuitable in an environment where high performance is required, or where computation and memory resources are scarce. Here, performance and memory consumption are critical aspects, that can lead to unacceptably long execution times. To solve this problem it is important to find out where the bottlenecks are and thus where the biggest room for improvement lies to develop new methods with a high optimization potential.

One important statistical learning application that requires a huge amount of resources is *Model-Based Optimization* (MBO), which can be used for parameter optimization of machine learning algorithms [HHL11]. Within a huge search space, algorithm configurations with different resource demands are evaluated and compared to find the configuration (model) with the best prediction quality. An important step towards better managing the huge search space is to parallelize the search process itself. However, doing this in a naive fashion that disregards the specifics of the underlying parallel architecture or uses inefficient scheduling strategies may lead to wasting resources.

This dissertation addresses methods for efficient resource utilization of statistical machine learning applications. Regardless of whether we are looking at making a single, specific machine learning algorithm faster or looking at an efficient parallelization strategy for MBO, the motivation remains the same: being able to tackle larger problem sizes without increasing the resource utilization or end-to-end runtime, or, alternatively, reducing the resource utilization and runtime while the problem size remains constant.

1.2.1 The Domain Specific Language R

R is an open source programming language, used as the de facto standard software environment for the development of statistical applications. Its main characteristic is a large set of dynamic features which allow the rapid development of new algorithms. The Comprehensive R Archive Network (CRAN)[1] includes over 11,000 software

[1] CRAN: http://cran.r-project.org, 2018

packages for statistical computing and the Bioconductor[2] repository contains over 1,300 packages for the analysis of genomic data.

However, the flexibility comes at a price: R is considered to be a rather slow language consuming a huge amount of memory during execution [MHO+12], which is a critical aspect for computation intensive applications like machine learning algorithms. One of the reasons for this is the fact that R is an interpreted language, stemming from the fact that classical ahead-of-time compilation of R code is challenging due to its dynamic nature. To overcome this, some developers make use of the fact that R can interface with other languages such as C, C++ or Fortran [R C18b], to improve the performance of computation-intensive parts of the algorithm. This in turn places a higher burden on the developers who are forced to forgo the convenience offered by the R language that made it so popular in the first place. It also means they cannot rely on functions from other R packages for these parts.

While alternative languages for statistical computation have been proposed [Tie18; RIn18], the statistics community has shown no interest in these in spite of R's performance issues. The main reason for this is the vast amount of open-source software packages available in R. In recent years, multiple approaches have been developed to optimize the runtime of R applications. There are projects with the goal to create alternative, more efficient R implementations [Ber18; TDH12; KMM+14; KE18]. However, most of these are experimental, and the user base remains fairly small due to compatibility problems with the available R software packages. Other projects [WWP14; Nea18] attempt to provide a faster R by modifying the original GNU R instead of re-implementing it, in order to stay compatible with the available R packages. For all of these projects, their authors have shown improvements for simple R programs. However, whether these speed-ups translate to complex real-world applications like learning algorithms is a different matter.

Optimizations for a faster execution and also for an *efficient resource utilization* of statistical learning algorithms based on R can only be profitable if they cover two aspects: They need to *preserve compatibility* with the available software packages many R programs are based on and simultaneously *cover the real resource bottlenecks* of those algorithms. As a first step, for guiding optimization efforts in the appropriate direction, it therefore is indispensable to analyze the runtime and memory consumption characteristics of learning algorithms. With the analysis of bottlenecks arising during execution, the optimization potential for resource utilization can be estimated and new efficient optimizations can be developed. The R execution environment already includes profiling tools such as Rprof [R C18d] for analyzing bottlenecks, but the analysis is restricted to high-level characteristics only. Internal functions like memory management of the R Interpreter itself are outside of the scope of what can be profiled.

For a more precise analysis, new *profiling mechanisms* need to be developed to analyze runtime and memory behavior of statistical machine learning algorithms.

[2] Bioconductor: http://www.bioconductor.org, 2018

To fully benefit from *now-ubiquitous multi-core architectures*, it is essential that the analysis is not restricted to single-threaded algorithms but also includes parallel (typically "embarrassingly parallel") learning algorithms and their mapping and scheduling on underlying parallel architectures.

1.2.2 Model-Based Optimization

One important parallel learning application with huge resource demands is MBO, also known as Bayesian optimization. MBO is, for example, applied for automated *hyperparameter optimization* of machine learning algorithms [HHL11]. It is a state-of-the-art technique for efficient *black-box optimization* [JSW98]. A black-box function is an unknown function f mapping elements from a d-dimensional space to some (typically real-valued) quality metric. The goal is to find the global minimum (or maximum) of the function in a limited time. For hyperparameter optimization, the goal is to find the algorithm configuration with the best performance like prediction quality for classification algorithms. Different algorithms with different parameters are evaluated to find a well-performing configuration for the given data. Due to the huge model space, a large amount of resources is needed to evaluate the configurations [CVB+02]. In fact, the evaluation of a single configuration can take *several hours* [VH04].

MBO is a global optimization method that is not only applied in statistics for the optimization of machine learning hyperparameters but also for many other research fields where expensive models have to be optimized to find a well-performing configuration. To reduce the number of necessary evaluations of the black-box function, MBO uses a regression model to approximate the objective function. The processing cycle of MBO is shown in Figure 1.1.

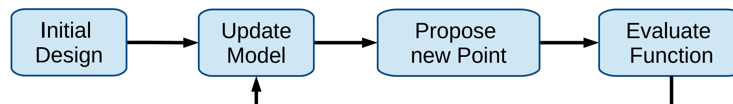

Figure 1.1: Simplified Visualization of the Model-Based Optimization Procedure.

Starting on an initial design of already evaluated configurations, the regression model guides the search to new configurations by predicting the outcome of the black-box on yet unseen configurations (response surface). Based on this prediction, an *infill criterion* (also called acquisition function) proposes a new promising configuration for evaluation. In each iteration, the regression model is updated on evaluated configurations of all previous iterations until the budget is exhausted.

Originally, the MBO algorithm sequentially proposes one configuration to be evaluated after another. To allow for *parallelization*, several modifications to the general technique or to the infill criteria have been suggested [HVC16] (such as constant liar, Kriging believer, qEI [GLC10], qLCB [HHL12], MOI-MBO [BWB+14]), that result in multiple points being proposed in each iteration. The number of

proposed configurations is typically chosen to equal the number of available CPUs. However, due to the *heterogeneous resource requirements* (CPU, memory etc.) for evaluating the different configurations, this can lead to inefficient resource utilization: Before new configurations can be proposed, the results of all evaluations within one iteration need to be gathered to update the model. The slowest evaluation thus becomes the bottleneck, and all other parallel worker processes idle after finishing their evaluation before a new iteration can start. One approach to avoid idling is to desynchronize the model update. Such asynchronous techniques have been suggested and discussed by [GJL11; JLG11; JLG+12]. Here, the main problem is to avoid evaluations of very similar configurations. Since the evaluation of a configuration can take several hours, the evaluation of similar configurations with little or no impact on the overall optimization is a waste of resources.

The overarching goal is to execute model-based optimization in a resource efficient way to enable the processing of larger problem sizes within a given time budget or, in other words, reduce the end-to-end wall-clock time for a constant problem size. This calls for the development of new *resource-aware scheduling strategies* to efficiently map configurations to the underlying parallel architecture, depending on their resource demands. In contrast to classical scheduling problems, the scheduling for MBO needs to interact with the configuration proposal mechanism to select configurations with suitable resource demands for parallel evaluation, which is a complex problem, since the resource demands need to be known (at least estimated) before execution.

1.3 Contribution of this Thesis

This thesis presents multiple methods for achieving efficient resource utilization for statistical machine learning applications.

One objective is to analyze the resource utilization of statistical learning applications implemented in the R programming language and develop optimizations that enable these algorithms to scale to larger problem sizes. As a first step towards this goal, the most common classification algorithms are analyzed with respect to their resource requirements, to determine where the highest optimization potential lies. This is enabled by enhancing and redesigning the *R profiling framework traceR*. Even if the analysis is focusing on learning algorithms, the results also support the development of new optimizations for the R programming language in general and also optimizations for alternative R implementations. The results show that one of the major contributors to the runtime issues is the memory consumption behavior.

To reduce the memory overhead of machine learning applications, a new optimization based on dynamic sharing of memory is developed. This optimization avoids duplication of page contents for large data structures and optimizes the copy-on-write mechanism of the R language. Its evaluation is based on different benchmark sets also including classification algorithms. Especially when the OS

starts to swap out memory the new optimization is able to speed up computation by several orders of magnitude.

In addition to the above-described new memory optimization, parallelization of the execution is used to speed up computation, which poses new resource utilization challenges. To analyze the bottlenecks arising in embarrassingly parallel machine learning applications like hyperparameter optimization, the traceR profiling framework is extended for the analysis of parallel programs. For hyperparameter optimization, the analysis with traceR shows that a high runtime variance in the configuration space can cause inefficient resource utilization due to different completion times of evaluations running in parallel. The results produced by traceR are used to develop new mapping and scheduling methods with respect to the resource utilization.

To enable a resource-efficient parallel variant of model-based optimization, a *Resource-Aware Model-Based Optimization* framework called RAMBO is developed and evaluated. Here, different resource-aware scheduling strategies are included. With RAMBO, it becomes possible to make use of the full potential of parallel architectures in an efficient manner. Therefore, a runtime estimation model that estimates the runtime for each evaluation of a configuration is developed to guide the mapping of evaluations to available resources. In addition to the runtime estimates, the scheduling strategies use an execution priority reflecting the estimated profit of an evaluation for finding the best configuration.

Different experimental setups are used to evaluate the performance of the RAMBO framework on heterogeneous and homogeneous parallel architectures. The results show that RAMBO manages to balance long execution times more evenly and thus executes more evaluations in the same time budget, leading to a higher confidence in the optimization space compared to the default parallel execution model. Compared to other parallel execution models for model-based optimization that aim at reducing the idle time by asynchronously updating the model, RAMBO converges faster to the optimum under the assumption that the resource estimates are accurate. In addition, the results for the scheduling strategy designed for heterogeneous architectures show that RAMBO converges faster to the optimum while consuming the same amount of energy compared to the competing approach.

1.4 Outline of this Thesis

The organization of the remaining parts of this thesis including the main contributions is visualized in Figure 1.2.

- **Chapter 2** provides an overview of the fundamentals of model-based optimization that will be optimized within Chapter 6. Furthermore, the fundamentals of the R language environment are presented, which builds the basis for the optimization of the resource utilization of learning algorithms in Chapter 4.

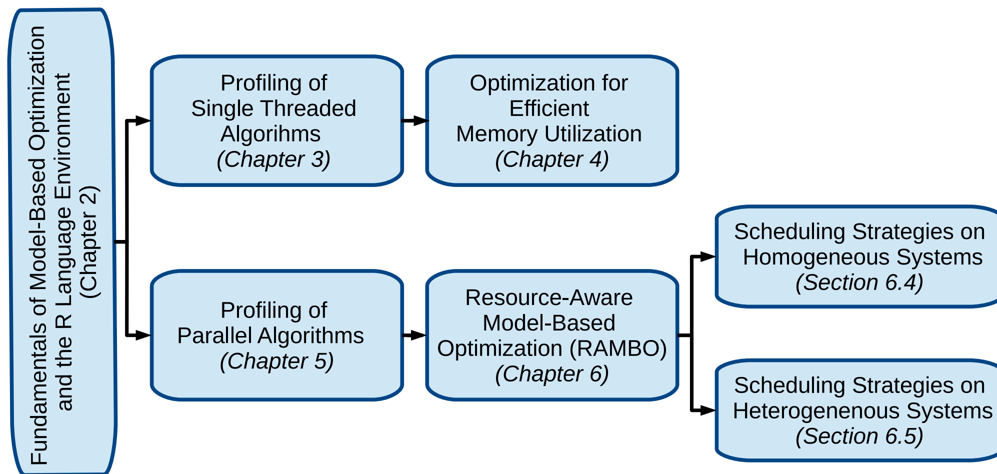

Figure 1.2: Organization and Contributions of the Thesis.

- **Chapter 3** starts with the related work on performance optimizations for R. It covers one of the main contributions of this thesis by presenting the analysis of the resource utilization of statistical learning algorithms implemented in the R programming language to support the development of new optimizations that enable these algorithms to scale to larger problem sizes. To accomplish this, a profiling framework was redesigned and enhanced.

- **Chapter 4** picks up the main results of the analysis presented in Chapter 3, leading to the second main contribution of this thesis: an optimization approach for efficient memory utilization of machine learning algorithms. Besides an extensive evaluation of this optimization, this chapter also contains a survey of related optimization approaches as well as the fundamentals of R's memory management system.

- In addition to the memory optimization in Chapter 4, the second major avenue for optimizing learning algorithms that is explored in this thesis is parallelization. **Chapter 5**, therefore, presents the analysis of parallel machine learning algorithms with the goal to develop new resource-aware scheduling methods. It furthermore includes the fundamentals of parallel execution models supported by the R programming language and presents the parallel profiling mechanisms of the profiling framework that was developed for the analysis.

- **Chapter 6** picks up the results presented in Chapter 5. It contains one of the main contributions of this thesis, a new resource-aware model-based optimization framework (RAMBO) including scheduling strategies for parallel MBO. The chapter starts by introducing the related approaches for the parallel execution of MBO and continues with presenting RAMBO and the

resource-aware scheduling strategies that can be applied on heterogeneous and homogeneous system architectures. The strategies are evaluated extensively and compared against existing approaches.

- **Chapter 7** summarizes this thesis and provides an outlook on future work.

1.5 Author's Contribution of this Thesis

According to §10(2) of "Promotionsordnung der Fakultät für Informatik der Technischen Universität Dortmund vom 29.August 2011", a dissertation within the context of doctoral studies has to contain a separate list that highlights the author's contributions to research and results obtained in cooperation with other researchers. For this purpose, the author's contribution to publications which lead to the contents of chapters 3, 4, 5 and 6 are described in the following:

- **Chapter 3:** In this chapter, an analysis of the resource utilization of statistical learning algorithms implemented in the R programming language is presented based on publications [KKL+14] and [KKK+14]. Those publications were mainly written by the author of this thesis. The evaluation results were performed in cooperation with Ingo Korb. The ideas and concepts of the analysis were mainly developed by the author of this thesis and partly evolved in discussions among co-authors.
  The benchmarks [KL18] were designed in collaboration with Michel Lang, who also carried out their implementation. The traceR tool [tra18] used for the analysis, was originally developed by Morandat et al. [MHO+12]. An initial redesigned and enhanced version was developed by the author of this thesis. An improved version was later implemented by Ingo Korb.
  Furthermore, the publication [KPM12] was referenced as a related approach. It was mainly written by the author of this thesis, while the concepts evolved in discussions among the co-authors.

- **Chapter 4:** This chapter is mainly based on publication [KKE+14] presenting an optimization approach for efficient memory utilization for learning algorithms. Most sections of the publication were written by the author of this thesis. The original idea for the optimization came from the author of this thesis based on the results from [KKL+14] and was refined in discussions among all co-authors. The implementation was carried out by Ingo Korb.
  Furthermore, the publication [KKM16] was referenced as related approach. Ingo Korb was the main author while the author of this thesis contributed with the idea and concept of the evaluation part.

- **Chapter 5:** In this chapter an analysis of parallel learning algorithms based on publications [KKM15a] and [KKM15b] is presented. The author of this

thesis was the lead author of these publications. The evaluation results were generated in cooperation with the co-author Ingo Korb, who implemented the parallel extension of the traceR tool [tra18]. The other co-authors contributed through technical discussions.

- **Chapter 6:** This chapter presents resource-aware scheduling strategies for parallel MBO and is based on publications [RKB+16], [KRL+16], [KRL+17] and [KLN+17]. The general idea of optimizing MBO via resource-aware scheduling came from Peter Marwedel. The main idea of the RAMBO framework and its first evaluation including the first fit scheduling strategy were carried out in [RKB+16]. Here, Jakob Richter and the author of this thesis contributed equally. The concept of the RAMBO framework was entirely designed by the author of this thesis, while Jakob Richter was responsible for the evaluation part. The RAMBO framework is based on the mlrMBO library [BBH+14] that was implemented by a multitude of people, among others Jakob Richter, Michel Lang, Bernd Bischl and Janek Thomas.
  In [KRL+17] and [KRL+16], an enhanced knapsack based scheduling strategy and a comparison study with several other parallel MBO approaches was presented. The idea and concept of the scheduling algorithm and the comparison study were mainly designed by the author. The scheduling algorithm was later implemented in cooperation with the author's bachelor student Andreas Lang. The parallel MBO approaches that RAMBO was compared to were described and implemented by Jakob Richter, Janek Thomas, Michel Lang and Bernd Bischl. They and the other co-authors also contributed through technical discussions. The first generation of evaluation results was done in cooperation with Jakob Richter and later largely re-worked and analyzed by the author of this thesis.
  The RAMBO approach for heterogeneous systems [KLN+17] was written and developed by the author of this thesis and later implemented in cooperation with Andreas Lang, based on technical discussion among all co-authors. The energy measurements were performed with a tool provided by Olaf Neugebauer [Neu17], who also advised the author on carrying them out.

1.6 Publications Covered by this Thesis

Parts of this thesis have been published as a technical report as well as in journals and proceedings of the following conferences.

- Helena Kotthaus, Ingo Korb, Michel Lang, Bernd Bischl, Jörg Rahnenführer, Peter Marwedel: *Runtime and Memory Consumption Analyses for Machine Learning R Programs.* Journal of Statistical Computation and Simulation, vol. 85.1, pp. 14-29, 2014

- Helena Kotthaus, Ingo Korb, Michael Engel and Peter Marwedel: *Dynamic Page Sharing Optimization for the R Language.* In Proceedings of the 10th Symposium on Dynamic Languages, pp. 79-90, Portland, USA, 2014

- Helena Kotthaus, Ingo Korb and Peter Marwedel: *Performance Analysis for Parallel R Programs: Towards Efficient Resource Utilization.* Technical Report 01/2015, Department of Computer Science 12, TU Dortmund University, SFB876 Project A3, 2015

- Ingo Korb, Helena Kotthaus and Peter Marwedel: *mmapcopy: Efficient Memory Footprint Reduction using Application Knowledge.* In Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp. 1832-1837, Pisa, Italy, 2016

- Jakob Richter, Helena Kotthaus, Bernd Bischl, Peter Marwedel, Jörg Rahnenführer and Michel Lang: *Faster Model-Based Optimization through Resource-Aware Scheduling Strategies.* In Proceedings of the 10th International Conference on Learning and Intelligent Optimization (LION 10), vol. 10079 of Lecture Notes in Computer Science, pp. 267-273, 2016

- Helena Kotthaus, Jakob Richter, Andreas Lang, Janek Thomas, Bernd Bischl, Peter Marwedel, Jörg Rahnenführer and Michel Lang: *RAMBO: Resource-Aware Model-Based Optimization with Scheduling for Heterogeneous Runtimes and a Comparison with Asynchronous Model-Based Optimization.* In Proceedings of the 11th International Conference on Learning and Intelligent Optimization (LION 11), vol. 10556 of Lecture Notes in Computer Science, pp. 180-195, 2017

Parts of this thesis have been published as short abstracts in the following conferences:

- Helena Kotthaus, Sascha Plazar and Peter Marwedel: *A JVM-based Compiler Strategy for the R Language.* In Abstract Booklet at the International R User Conference (UseR!) WiP, page 68, Nashville, Tennessee, USA, 2012

- Helena Kotthaus, Ingo Korb, Markus Künne and Peter Marwedel: *Performance Analysis for R: Towards a Faster R Interpreter.* In Abstract Booklet of the International R User Conference (UseR!), page 104, USA, Los Angeles, 2014

- Helena Kotthaus, Ingo Korb and Peter Marwedel: *Performance Analysis for Parallel R Programs: Towards Efficient Resource Utilization.* In Abstract Booklet of the International R User Conference (UseR!), page 66, Aalborg, Denmark, 2015

- Helena Kotthaus, Jakob Richter, Andreas Lang, Michel Lang and Peter Marwedel: *Resource-Aware Scheduling Strategies for Parallel Machine Learning R Programs through RAMBO.* In Abstract Booklet of the International R User Conference (UseR!), page 195, USA, Stanford, 2016

- Helena Kotthaus, Andreas Lang, Olaf Neugebauer and Peter Marwedel: *R goes Mobile: Efficient Scheduling for Parallel R Programs on Heterogeneous Embedded Systems.* In Abstract Booklet of the International R User Conference (UseR!), page 74, Brussels, Belgium, 2017

Chapter 2

Fundamentals: Model-Based Optimization and the R Language

This thesis addresses methods for efficient resource utilization of statistical machine learning applications. For this purpose, this chapter provides an overview of the fundamentals of model-based optimization, as a machine learning algorithm with huge resource demands, and the GNU R programming language as the de facto standard software environment for the development of statistical learning applications.

R is an open source programming language. Its main characteristic is a large set of dynamic features which allow for the rapid development of new algorithms. The *Comprehensive R Archive Network* (CRAN) supports the R ecosystem with over 12,000 R software packages for statistical computing, and the *Bioconductor* repository with over 1,300 R packages for the analysis of genomic data. Due to the high amount of available software packages R is not only used as a programming language in statistics, it is also a popular graphics tool and data analysis platform in various areas like finance, biology or physics. As reported by Rexer Analytics, which is the largest survey of data science in industry, R has gained high popularity in recent years. While in 2007 only 23% of the respondents used R, in 2015 already 76% of them were using R and more than a third use it as their primary data analysis platform[1]. R is therefore also popular in the domain of machine learning.

R offers a wide variety of machine learning algorithms via packages[2]. Additionally, frameworks that offer interfaces to several machine learning algorithms are available like `caret` [Kuh08], `mlr` [BLK+16], `Rweka` [HBZ09] or `h2O` [Kt17]. Most of the machine learning algorithms allow the adjustment of their parameters. The parameter configuration of an algorithm can highly influence its prediction performance and also its resource utilization. Thus, it is important to adjust those parameters, which require deep knowledge of the algorithms or, alternatively, an automatic hyperparameter optimization approach [THH+13; LKM+15]. The *hyperparameters* are the parameters of an algorithm that are configured before the learning algorithm starts.

[1] Rexer Analytics: http://www.rexeranalytics.com/data-science-survey.html, 2018

[2] CRAN Task View: Machine Learning & Statistical Learning: https://cran.r-project.org/web/views/MachineLearning.html, 2018

For automated hyperparameter optimization, different approaches can be applied. One popular global optimization method is *Model-Based Optimization* (MBO) [HHL11]. R also offers a toolbox for MBO called `mlrMBO` [BBH+14; BRB+17]. MBO is a state-of-the-art technique for efficient optimization of *expensive black-box functions* [JSW98], it is also known as Bayesian optimization. A black-box function is an abstraction of a system for which the internal working is unknown. For hyperparameter optimization the machine learning algorithm represents the black-box function. To reduce the number of necessary evaluations of algorithm configurations, MBO uses a regression model to approximate the outcome of the black-box. The goal is to find the configuration with the highest quality of the output measured by a given performance criterion within a limited time budget. Due to huge model spaces (many different parameter configurations), a large amount of resources is needed to evaluate these configurations [CVB+02; VH04]. MBO, as an important machine learning algorithm with huge resource demands, is therefore addressed within this thesis.

The remainder of this chapter is structured as follows: Section 2.1 presents the fundamentals of MBO including hyperparameter optimization. An overview of the R language environment including the R language characteristics and execution model is given in Section 2.2. Section 2.3 summarizes this chapter.

2.1 Model-Based Optimization

In recent years, statistical machine learning has emerged as a key technique for tackling problems that elude a classic algorithmic approach. One such problem, with a major impact on human life, is the analysis of complex biomedical data. Different machine learning algorithms are utilized like classification for predicting the health status of a patient or MBO for finding the best algorithm configuration for the prediction. While achieving the highest possible prediction quality appears desirable, it is often simply infeasible due to time and budget constraints. This applies especially in the field of personalized medicine, where the patient needs a treatment as soon as possible.

The hyperparameters of an algorithm can heavily affect both its prediction accuracy as well as its resource utilization. Therefore it is important to tune these parameters. Different approaches can be applied. MBO is one popular method used for hyperparameter optimization, that tries to minimize the number of necessary configuration evaluations to find the best algorithm configuration as soon as possible.

In this section, first the concept of hyperparameter optimization is presented in Subsection 2.1.1 and then the MBO algorithm as one method for automated hyperparameter optimization is introduced in Subsection 2.1.2.

2.1.1 Hyperparameter Optimization

The hyperparameter configuration of a machine learning algorithm highly affects its prediction performance and also its resource utilization like its runtime. Dependent on the algorithm and its parameters a single evaluation of a configuration can take up to several hours [VH04].

A manual optimization of these parameters would require expert knowledge of the implementation and is often infeasible due to the huge model space that spans over all possible parameter configurations. Furthermore, the performance of an algorithm and thus its optimal configuration can vary dependent on the input data set.

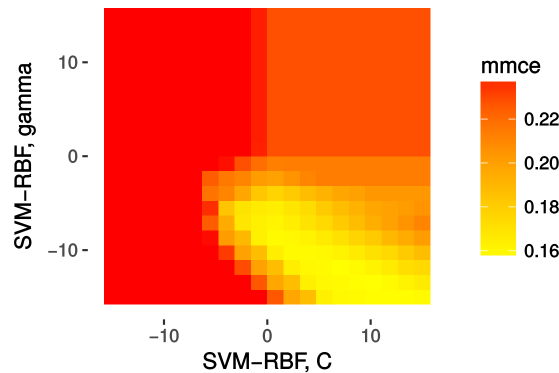

Figure 2.1: Visualization of the mean missclassification error (mmce) as a performance measurement for different configurations of a SVM classification task with radial basis function kernel (SVM-RBF).

Figure 2.1 visualizes the heterogeneity of the performance of different configurations of a *Support Vector Machine* (SVM) with a *Radial Basis Function* kernel (RBF) used for classification. Dependent on how the parameter for the kernel γ on the x-axis and the cost parameter C of constraint violations on the y-axis are chosen, different mean misclassification errors (mmce) are generated [KBF+12]. Here, the goal of hyperparameter optimization is to find the configuration with the lowest mmce (light yellow) within a limited time budget.

In general, the hyperparameter optimization problem can be formulated as follows: Given an algorithm A that is parameterized and problem instances I and a cost metric c, find the parameter configuration θ^* that minimizes (or maximizes) c on I [HHL11]. For machine learning algorithms, the cost metric c quantifies the quality of the configuration like, e.g., the classification performance that can be evaluated via different resampling methods like, e.g., cross-validation [GWH+13].

The space of possible configurations Θ forms the search space for the optimization problem:

$$\theta^* := \text{argmin}_{\theta \in \Theta} f(\theta). \tag{2.1}$$

Here, $f(\theta)$ denotes the evaluation of the algorithm with θ as parameter configuration and can be interpreted as the evaluation of a black-box function $f : \mathcal{X} \to \mathbb{R}$, were the parameter configurations form the input and the quality (e.g., prediction quality) forms the output (see Subsection 2.1.2).

For automated hyperparameter optimization, several methods were developed, e.g., the random search methods [BB12], evolutionary algorithms [LSS12; AST09], meta-learning approaches [BGS+08], bandit-based methods [LJD+17], racing algorithms [BBS07; LKM+15] or Bayesian optimization approaches like MBO [SLA12; KFB+17]. Additionally, combinations of these methods can be applied, e.g., Hutter et al. [FKH17] proposed a combination of bandit-based methods that are based on the random search with MBO for hyperparameter optimization.

This thesis will solely focus on MBO approaches for hyperparameter optimization, since MBO is one of the most efficient methods for reducing the number of necessary evaluations of configurations [BRB+17; Jon01].

2.1.2 Model-Based Optimization Algorithm

The MBO algorithm is not only applied in statistics for the optimization of hyperparameters, but also in many other research fields where expensive models have to be optimized to find a well-performing configuration. MBO is a state-of-the-art algorithm for expensive black-box functions in the field of Design and Analysis of Computer Experiments (DACE) [SWM+89].

The challenge of this global optimization method is to find the best configuration possible or so-called global optimum of a given black-box function $f : \mathcal{X} \to \mathbb{R}$, $f(\boldsymbol{x}) = y, \boldsymbol{x} = (x_1, \ldots, x_d)^T$ with a d-dimensional input domain $\mathcal{X} \subset \mathbb{R}^d$ and an output y. Here, f is usually expensive to evaluate and thus the number of configuration evaluations is limited by a time budget. It is furthermore assumed that $\mathcal{X} \subset \mathbb{R}^d$ is expressed by simple box constraints. The values of $\mathcal{X}$ can not only be numeric but also categorical. The goal is to find the input $\boldsymbol{x}^*$ (former θ^* see Equation (2.1)) and thus the best configuration of the black-box with:

$$\boldsymbol{x}^* := \text{argmin}_{\boldsymbol{x} \in \mathcal{X}} f(\boldsymbol{x}). \tag{2.2}$$

Sequential Mode-Based Optimization (SMBO) Algorithm: The general Sequential MBO (SMBO) algorithm is visualized in Figure 2.2. Starting on an *initial model* with already evaluated configurations of f, a regression model $\hat{f}$ is fitted. The initial set can be generated with different sampling methods described in [MBC00]. The *regression model* tries to describe the dependencies between the configuration of the algorithm (input) and the quality of its output (response) measured by a given

performance criterion. The evaluated configurations of the initial set are usually chosen in a space-filling manner to uniformly cover the input space.

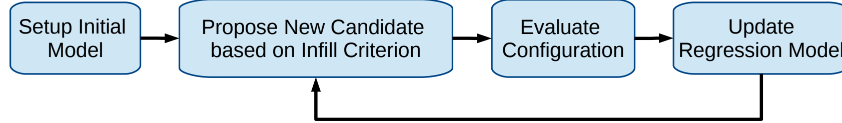

Figure 2.2: Visualization of the Sequential MBO (SMBO) Algorithm.

The regression model guides the optimization by estimating its response surface. The response surface represents the estimated outcome of the black-box function on yet unseen configurations. A *surrogate model* is often used as a regression model, as it is comparably inexpensive to evaluate and therefore often used when function evaluations are very expensive [FSK08]. It iteratively proposes a new promising configuration $\boldsymbol{x}$ that is determined by optimizing a so-called infill criterion (also called acquisition function).

The *infill criterion* quantifies the improvement of a configuration based on a compromise between good predicted outputs and uncertainty about the search space region (a high potential to optimize the quality of the regression model). To obtain the objective value y, $f(\boldsymbol{x})$ is evaluated and the surrogate is refitted and a new configuration is proposed for the next iteration. This process is repeated until a target objective value is reached or a predefined budget is exhausted.

SMBO Example: Figure 2.3 visualizes two exemplary MBO iterations steps. Here, y (solid line) in the upper parts of both MBO iteration steps denotes the output of the unknown black-box function f while $\hat{\boldsymbol{y}}$ (dotted line) denotes the outputs of the surrogate regression model $\hat{f}$ that tries to approximate the black-box function. The gray area around $\hat{\boldsymbol{y}}$ represents the uncertainty of the surrogate model.

In the *first MBO iteration* (see Figure 2.3, upper part) the initial set of configurations (red dots) are already evaluated. Based on the corresponding minimum of the infill criterion in the lower part of the MBO iteration figures (dotted line, lcb) a new configuration (blue triangle upper part) is proposed for evaluation.

In the *second MBO iteration* (see Figure 2.3, lower part) $\hat{f}$ is refitted with the evaluated configuration (green rectangle) and a new configuration is proposed (blue triangle) based on the infill criterion (lcb). This process continues until the budget is exhausted. This popular *Efficient Global Optimization* (EGO) algorithm was proposed by Jones et al. [JSW98].

Surrogate Model: EGO sequentially proposes one point to be evaluated after another using Kriging as a surrogate and the *Expected Improvement* (EI) as an infill criterion (2.3). *Kriging* (also called Gaussian process regression) is one of the most popular and flexible variants of a surrogate model [RW05; JSW98]. It is recommended

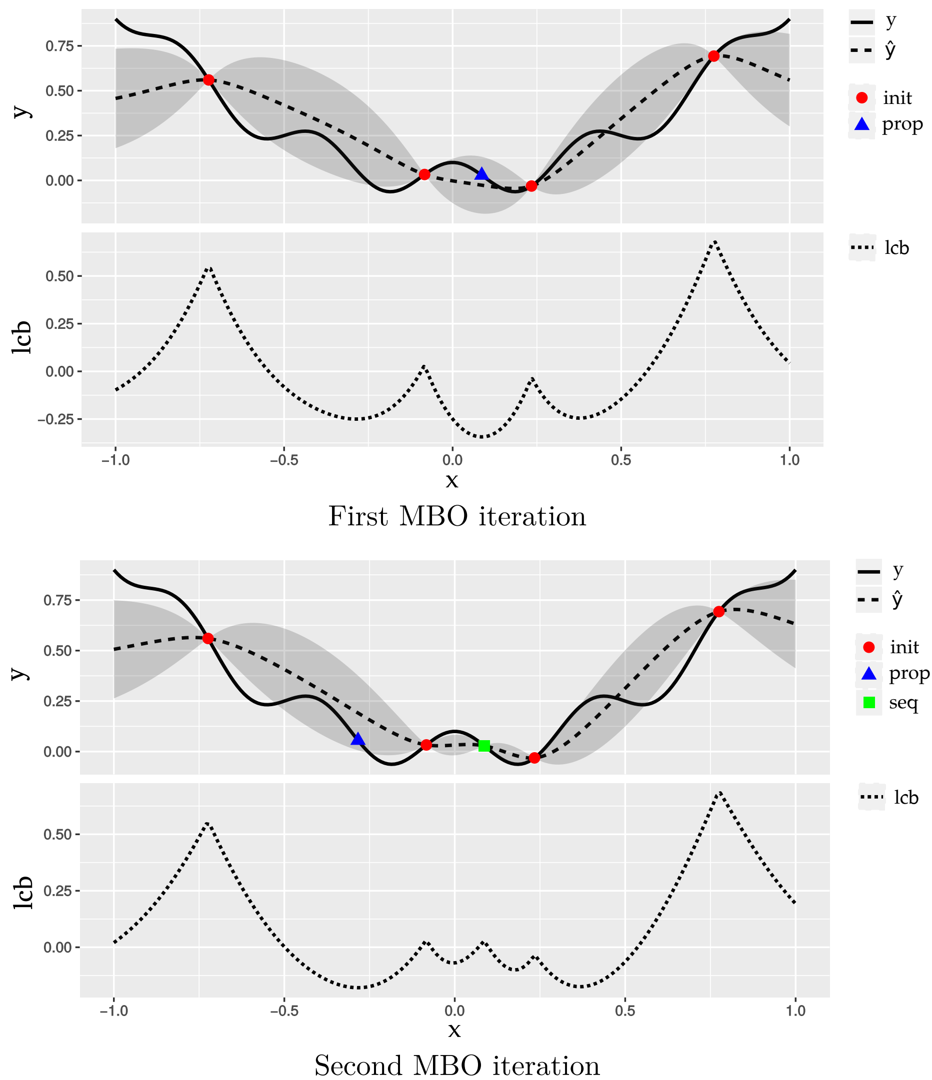

Figure 2.3: Visualization of two exemplary SMBO iterations with the *Lower-Confidence Bound* (LCB) as infill criterion.

when the input domain is numeric $\mathcal{X} \subset \mathbb{R}^d$. If $\mathcal{X}$ contains, e.g., categorical variables, the Random Forest algorithm is commonly used as surrogate [HHL11].

Infill Criterion: The infill criterion guides the optimization by quantifying the improvement of a configuration $\boldsymbol{x}$ based on a compromise between good predicted outputs $\hat{\mu}(\boldsymbol{x})$ (exploitation) and uncertainty about the search space region $\hat{s}(\boldsymbol{x})$ (exploration). The former described surrogate model based on Kriging is often combined with the popular EI criterion or with the *Lower-Confidence Bound* (LCB) criterion.

The EI criterion is defined as follows:

$$\begin{aligned}\mathrm{EI}(\boldsymbol{x}) &= \mathbb{E}(\max(y_{\min} - \hat{\mu}(\boldsymbol{x}), 0)) \\ &= (y_{\min} - \hat{\mu}(\boldsymbol{x}))\,\Phi\left(\frac{y_{\min} - \hat{\mu}(\boldsymbol{x}))}{\hat{s}(\boldsymbol{x})}\right) + \hat{s}(\boldsymbol{x})\phi\left(\frac{y_{\min} - \hat{\mu}(\boldsymbol{x})}{\hat{s}(\boldsymbol{x})}\right).\end{aligned} \tag{2.3}$$

Here, Φ is the distribution, ϕ is the density function of the standard normal distribution and $y_{\min}$ is the best observed objective value so far.

Alternatively, the comparably simpler LCB infill criterion can be utilized:

$$\mathrm{LCB}(\boldsymbol{x}, \lambda) = \hat{\mu}(\boldsymbol{x}) - \lambda\hat{s}(\boldsymbol{x}), \quad \lambda \in \mathbb{R}, \tag{2.4}$$

where $\hat{\mu}(\boldsymbol{x})$ denotes the posterior mean and $\hat{s}(\boldsymbol{x})$ the posterior standard deviation of the surrogate model for configuration $\boldsymbol{x}$. Exploitation and exploration are balanced, a good (low) expected value of the solution $\hat{\mu}(\boldsymbol{x})$ is rewarded, and high estimated uncertainty $\hat{s}(\boldsymbol{x})$ is penalized. The λ variable guides the exploration-exploitation trade-off.

In addition to the presented infill criteria, several other infill criteria [Jon01], specializations, e.g., for categorical search spaces like in the SMAC Framework [HHL11] and noisy optimization [RGD12] have been introduced. In this thesis, we will focus on the EI and LCB criterion.

Both EI and LCB are very popular and can be used for SMBO, where only one point is proposed per MBO iteration. To allow for parallelization within one MBO iteration, multiple points need to be proposed per iteration. To accomplish this, infill criteria based on the defined EI and LCB were suggested in [GLC10; HHL12] and will be explained in more detail in Section 6.1.

In the next section, the R language environment, which is used for the development of efficient model-based optimization within this thesis, will be presented.

2.2 The R Language Environment

The R language was designed by Ihaka and Gentleman and first released in 1993 [IG96]. It is influenced by two languages, the statistical programming language S [BC84] that was developed at Bell Laboratories in 1980 and the Scheme programming language [Dyb09]. Figure 2.4 visualizes the most important parts that the *R language environment* is built on and influenced by.

R is maintained by the *R Core Team* and the *R Foundation* of statistical computing and is available under the GPL license [R C18b]. R has also *commercial support* like the *TERR* environment, developed by TIBCO [F18] or *Microsoft R*, former Revolution R, provided by Revolution Analytics and is used and supported by many other global companies like Google, Oracle, Microsoft and IBM.

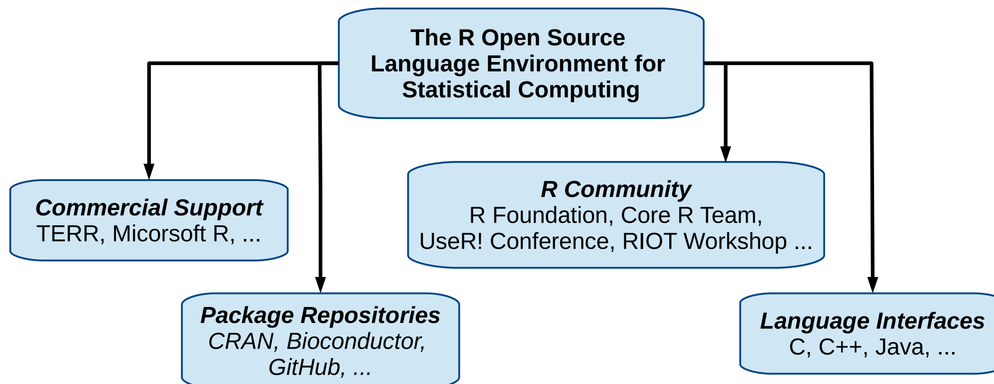

Figure 2.4: Categories that influence and build up the R language environment.

The R implementation includes about 700 built-in *packages*. Packages are libraries including R functions that can be dynamically loaded into an R program. Besides the built-in packages that contain the basic functions of R, R can be extended with over 13,000 packages, available on different repositories like CRAN[3], Bioconductor[4], and GitHub. The CRAN repository offers several task views where those packages are sorted by topics.

Once a year the R community meets at the *UseR!* conference to present and discuss their new packages and optimizations for R. Furthermore, since 2015 a group of researchers, including members of the Core R Team that concentrates on alternative R implementations and performance optimizations for R meet once a year at the *R Implementation, Optimization and Tooling Workshop* (RIOT).

Since R is highly extensible, it can *interface with other languages*, thus packages contain not only R code but also code from other languages like C, C++, or Fortran [R C18d]. The R language has so far no formal language specification. There exists only a first draft that was published at the end of 2017 [R C17b], the specification is only given by R's source code that is frequently changed.

The R Language is processed by an Interpreter that is mostly written in C code. In 2011 a bytecode compiler was added to the R language by Tierney [Tie01]. It provides the option to compile R code into a bytecode representation for faster evaluation. This enables optimizations which are generally beneficial for explicit loops in R code. R is commonly used interactively via a command-line interface, called *Read-Eval-Print-Loop* (REPL) that supports the rapid prototyping of statistical applications [R C18a]. The next section presents the language characteristics of R.

[3]CRAN: `http://cran.r-project.org`, 2018
[4]Bioconductor: `http://www.bioconductor.org`, 2018

2.2.1 Language Characteristics

R is a vector-based language with functional paradigms but also has object-oriented characteristic and is dynamically typed. This section is based on the draft of the R language specification [R C17b] and the analysis of the R languages characteristics by Morandat et al. [MHO+12].

Vector-based: In R everything is an object that is represented by a so-called S-Expression (symbolic expression) in the underlying C code representation [R C18c]. The S-Expression type describes how the contents of the object are treated, which is important for the execution by the interpreter. The basic data structure in R is a vector object. Vectors contain elements of the same type, e.g., primitive data types like, logical, integer, real, complex, character or raw.

The values of a vector can be accessed via indexing operations, e.g., $x[3]$ - returning the third element of vector x. Missing observations, which are commonly needed in statistics are represented by the value NA. Vectors that contain values of any type of an R object are called lists or generic vectors. R objects can contain attributes. Attributes are name-value pairs. For example, if a vector contains an attribute called “dim” (dimension) this vector represents a matrix.

Furthermore, R offers special compound objects called factors and data frames. *Factors* are used to categorize data and are stored as a vector of integers with a corresponding set of characters. *Data frames* are used for storing data tables. They consist of a generic vector of vectors, factors or matrices. R has also several other data types used for internal processing like environments, function objects, built-in objects, pairlists or promises (for further details see Chapter 3).

Functional paradigms: In R functions are treated as *first class objects*. They can be manipulated as any other R object, returned by other functions, and passed as arguments to other functions. Function objects consist of an argument list, a function body, and an environment and are thus closures. An environment is created when a function is called and contains references to variables that can be accessed and manipulated by the function. Thus functions are *lexically scoped* like in imperative programming languages. R also provides a global environment and environments for packages.

Each argument of a function is stored in a promise for *lazy evaluation*. A promise contains the reference to the lexical environment of the argument. R provides referential transparency by passing arguments by deep copy (pass-by-value semantics) except for environments that are passed by reference. However, side effects are possible, since the environment of a function that holds the variables, can be manipulated after its creation.

Dynamic characteristics and computing on the language: The R programming language has a dynamic type system, arguments that are passed to a function

are frequently type checked or converted inside the function. Furthermore, R allows for computing on the language. The user can create new language objects at runtime to store code that can then be passed around and executed via the so-called "eval" function. Environments can directly be created by the user. They can be attached to the search path or to a function. The concept of reflection allows the manipulation of variables inside environments including the call stack.

Object-oriented: R supports several object-oriented systems that are influenced by the functional characteristics of the language and thus different from languages like Java or C++. The two most popular ones are called S3 and S4 [Cha16].

The *S3 system* is based on so-called generic functions with single dispatch and has no formal definition of classes. Instead of the object, the generic function decides which method is called. An S3 object has an attribute attached called "class" attribute. This attribute is a vector of strings that is used to decide in which order methods are resolved. The generic functions take the S3 object as a parameter and look up the class attribute to dispatch the matching function from most to least specific. This system is commonly used in R and also applied for the internal parts of the interpreter to dispatch on built-in functions.

The *S4 system* also uses generic functions like S3, but introduces formal classes that define the inheritance structure instead of just using an attribute. Furthermore, it allows for multi-methods, meaning that the method dispatch supports multiple arguments to decide which method to call. The S4 system is complex and not that often used compared to the S3 system.

2.2.2 Execution Model

The R language is evaluated by an interpreter [R C17b]. This evaluation process is visualized in Figure 2.5.

The R *Parser* converts the textual R code expressions into an internal representation that represents an *Abstract-Syntax-Tree* (AST). The AST consists of R language objects like language expressions (Exp) and constant values (C) or symbols (S). It is traversed and executed by the R *Evaluator*. The evaluator returns the value (V) of the parsed expression.

Most language objects in this AST are stored as function calls that contain the function name and a list of arguments. Function objects are dispatched to the evaluation of their bodies together with their argument lists that are stored as promises for lazy evaluation. Each time a function is invoked a context is created and placed on a stack that is needed for error handling or control flow operations like the return operation. When the evaluation of the context has finished, it is removed from the stack [R C17b].

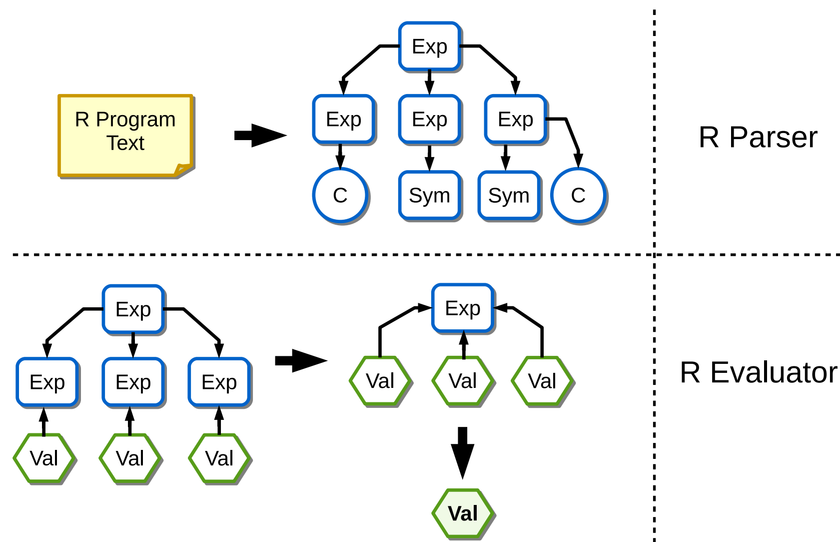

Figure 2.5: Execution Model of the R Language.

2.3 Summary

This chapter provided an overview on the fundamentals of this thesis including MBO - a machine learning algorithm with huge resource demands - and the R programming language - the de facto standard software environment for the development of statistical learning applications.

In the following chapters, different methods for efficient resource utilization in statistical machine learning algorithms based on the R programming language will be presented. While Chapter 3 and Chapter 4 focus on single-threaded machine learning applications and their resource bottlenecks induced by the R programming language, Chapter 5 and Chapter 6 will focus on parallel machine learning applications and the optimization of the parallel variant of the described MBO approach.

Chapter 3

Profiling of Machine Learning Algorithms

This chapter presents an analysis of the resource utilization of statistical learning algorithms implemented in the R programming language and is based on the papers by Kotthaus et. al. [KKL+14; KKK+14]. GNU R is the most widely used programming language for statistical data analysis in general, and biostatistics in particular. While apparently not affecting its popularity, its lavish use of resources makes it unsuitable in an environment where high performance is required, or where computation and memory resources are scarce. Here, runtime performance and memory consumption are critical aspects, that can lead to unacceptably long execution times. To solve this problem it is important to find out where the bottlenecks are and thus where the biggest room for improvement lies.

One major hurdle for efficient R programs is that classical ahead-of-time compilation of R code is hindered by R's highly dynamic nature. This leads some users to re-implementing performance critical parts of their algorithms in C or C++ to achieve a higher execution speed. However, translating bigger parts of an R program to another language is a complex task, since R programs rely on functions from R libraries or basic functions included in the R interpreter execution environment that might not be readily available in other environments.

In fact, one of the main drivers behind R's popularity is the vast amount of available open source software packages, a fact that has also proven to be a near-insurmountable obstacle for alternative statistical computation languages that have since been proposed [Tie18; RIn18]. In spite of R's well-known performance issues, these languages have never gained any significant traction within the statistics community. In recent years, multiple approaches have been developed to improve the execution speed of R applications. There are projects with the goal to create alternative, more efficient R implementations [Ber18; TDH12; KMM+14; KE18]. However, most of these are experimental, with only a few users due to compatibility problems with the available R software libraries. Other projects [WWP14; Nea18] attempt to provide a faster R by modifying the original GNU R to stay compatible with the available R libraries without the need of reimplementation. All of these projects have usually shown improvements for simple R programs and micro-benchmarks, but they exhibit fairly mixed results when it comes to speeding up complex real-world applications like machine learning algorithms.

Optimizations for a faster execution and also for an efficient resource utilization of statistical learning algorithms based on R can only be profitable if they cover two aspects: Staying compatible with the available software packages most R programs are based on, and simultaneously covering the real resource bottlenecks of those algorithms. It is therefore indispensable to analyze the runtime and memory consumption characteristics of learning algorithms. With the analysis of bottlenecks arising during execution, the optimization potential for resource utilization can be estimated and new efficient optimizations can be developed. The R execution environment already includes profiling tools such as Rprof for analyzing bottlenecks, but the analysis is restricted to high-level characteristics only. In particular, internal functions like memory management of the R Interpreter itself are outside of the scope of what can be profiled. For a precise analysis, new profiling mechanisms need to be developed to analyze runtime and memory behavior of real-world R programs.

The objective of this chapter is to analyze the resource utilization of statistical learning algorithms to support the development of new optimizations that enable these algorithms to scale to larger problem sizes. As a first step towards this goal, the most common classification algorithms are analyzed with respect to their resource requirements, to determine where the highest optimization potential lies. To accomplish this, an R profiling framework, called traceR [tra18], is redesigned and enhanced. Even if the analysis is focusing on learning algorithms, the results also support the development of new optimizations for the R programming language in general.

This chapter is structured as follows: First, Section 2.2 gives an overview of the R language environment including the R language characteristics and execution model of R. First, the related approaches of optimizations for the R language are presented in Section 3.1. Section 3.2 then describes the profiling framework that serves as a basis for the performance analysis. An overview of the machine learning benchmarks and their input data sets used in the analyses is given in Section 3.3, followed by a detailed analysis of their runtime and memory behavior. This analysis serves as a starting point for developing approaches to overcome the identified bottlenecks. Finally, the results are summarized in Section 3.4.

3.1 Optimizations for R - Existing Approaches

A major hurdle for general speedups of R programs is that R is executed by interpretation as opposed to it being compiled to machine code. Classic ahead-of-time compilation of R code is hindered by the fact that R is highly dynamic. Thus information like data types which is needed for optimizations in the compiler is only available at runtime. For example, when a function is declared in an R program, no data types need to be specified for the parameter list. When such a function is called, there are multiple ways to pass the same set of arguments. These features make R very convenient for the user, but very inconvenient for ahead-of-time compilation.

Other languages with a similarly dynamic nature like Matlab or Python have overcome such runtime issues by using *Just-In-Time* (JIT) based compilation approaches [AP01; BCF+09]. These approaches use knowledge gained at runtime to specifically compile fragments for the time-intensive parts instead of either compiling the entire program at once or interpreting the program statement-by-statement.

One popular runtime environment that provides a just-in-time compilation is the *Java Virtual Machine* (JVM) [LYB+14]. Kotthaus et al. [KPM12] presented a concept for implementing an optimized version of R by targeting the JVM. Existing alternative R execution environments that also target the JVM are the fastR project [KMM+14; SWH+16] and Renjin [Ber18]. But also other VM implementations were utilized to speed up R, e.g., the NQR project that targets the Parrot VM [KE18]. Furthermore, approaches that propose experimental specialized JIT compilers for R exist [TDH12; TDH14]. These approaches reimplement the original R interpreter that is written in C, in another language such as Java or C++ and benefit from optimizations available to their runtime environments. However, the reimplementations cannot yet guarantee full compatibility with existing R programs and libraries due to the complex and evolutionary development of the R language and its *missing formal specification*.

Other projects like pqR [Nea18] or Orbit VM [WWP14; WPW15] attempt to provide a faster R interpretation by modifying the original R interpreter instead of reimplementing GNU R to stay compatible. The original GNU R execution environment also contains the option to compile R functions into byte code for faster evaluation which provides some improvement in runtime especially for programs that use loops [Tie01]. Furthermore, additional libraries were developed to speed up arithmetic operations by taking advantage of specific processor architecture features like the Intel MKL [Int18] library or OpenBLAS [ZWW18]. Such libraries are optimized implementations of the reference BLAS (Basic Linear Algebra Subprograms) library that is included in the R execution environment.

All of the described optimization approaches have usually shown improvements for simple R programs or specific R functions, but they exhibit fairly mixed results when it comes to speeding up complex real-world applications like machine learning algorithms. One of the objectives of this thesis is to provide insights into the runtime and memory behavior of these algorithms on the original R execution environment. The hope is that both alternative R implementations as well as the original GNU R can use these results to develop optimizations that improve the runtime performance and resource utilization of real-world code.

Morandat et al. [MHO+12] already analyzed bottlenecks for R programs from different fields of statistics. Here, mostly artificial input data sets where used. However, as the characteristics of the input datasets vastly influence the runtime behavior of a program, only realistic data can yield results which are beneficial in practice. In this thesis, we focus specifically on machine learning algorithms combined with real-world input data sets from the UC Irvine machine learning repository (UCI) [BL18] in order to ensure a realistic scenario when analyzing

the main reasons for the lavish use of resources of R. Therefore, the R profiling framework traceR, presented in the next Section, is redesigned and enhanced.

3.2 Profiling Tool for R - traceR

The *traceR* framework, used for the profiling of machine learning algorithms in this thesis, was originally developed at the Purdue University [Rea12] for R version 2. As part of this thesis, traceR was reimplemented and redesigned with improved usability and analysis capabilities to provide more detailed insights, such as gathering more information about the sizes of vector data structures used during the execution of an R program. Besides, traceR was also ported to the current R version 3 and is available on GitHub [tra18]. In addition, traceR was also augmented with new profiling mechanisms to allow for profiling parallel R applications (see Chapter 5).

The R runtime environment already provides profiling tools like Rprof [R C18d] and several libraries like profviz [CLC+17] to visualize the profiling data generated by Rprof. However, in contrast to traceR, Rprof is a *sampling profiler*, meaning it stops the execution of a program at regular intervals to record which function is currently executed. The advantage of a sampling profiler is the relatively low measurement overhead, which however comes at the cost of a greatly reduced precision. In fact, the results can vary greatly between two profiling runs of the same problem with the same input data, especially when many functions with short execution times are involved. The runtime of functions with an isolated runtime smaller than the timing interval can be miscalculated since these function invocations might remain invisible to the profiler as they “fall through the cracks”. This may be true even if the cumulated runtime of such functions has a significant impact on the overall execution time of a program. In the worst case, the program consists of many short function runtimes that when summed up even dominate the runtime of the entire program but are mostly missed by the profiler.

Using Rprof furthermore restricts the analysis to high-level characteristics only; details about the internals of the R runtime environment functions are not provided. Also, the available memory profiling tools like Rprofmem [R C18d] are unsuitable for the purpose of a detailed memory behavior analysis – for example, memory allocations related to certain types of user data are reported, but internal data types like pairlists, which are used for passing arguments in a function call, are outside of the scope of what can be profiled.

For the development of new efficient optimization for R it is indispensable to have a detailed view into the internals of the R execution environment or so-called interpreter. The traceR profiling mechanisms are directly integrated with the R interpreter code, which enables the generation of more detailed data. For instance, data about how much of the program is spent in C/Fortran code supplied by R packages can be measured, or the runtime needed for memory management tasks like garbage collection can be analyzed. The traceR framework uses a *deterministic*

profiling approach, where each interesting location of the R interpreter is instrumented with an explicit call to the profiler. Thus, no function call can be missed, but a larger overhead is incurred compared to the sampling approach. To reduce the overhead, traceR is separated into two instrumented versions of the R interpreter, one to record runtime information and one to analyze non-time-related behavior like memory allocations or details about function parameters. This separation removes the overhead of the memory measurements from the runtime measurements.

The next section presents the benchmarks that were used for the analyses based on traceR.

3.3 Profiling of Machine Learning Algorithms Written in R

To determine where the highest potential for efficient optimizations of machine learning R algorithms lies, the runtime and memory behavior of these algorithms is analyzed in this section. Here, the most common classification algorithms are profiled with respect to their resource utilization. As the characteristics of the input data set can influence the runtime and memory behavior of an algorithm, real-world input data sets from the UCI repository [BL18] are used to ensure a realistic scenario when analyzing the main reasons for the lavish use of resources. Subsection 3.3.1 gives an overview of the machine learning benchmarks used in the analyses. The results for the runtime behavior are presented in Subsection 3.3.2 and the results for the memory behavior are described in Subsection 3.3.3.

3.3.1 Experimental Setup

For a thorough analysis, a large number of some of the most popular machine learning algorithms is used and applied on seven publicly available classification tasks as input data from the UCI repository [BL18]. The benchmark selection is based on the popularity and availability of the algorithm's implementations. The benchmarks are publicly available [KL18]. In the following all *classification algorithms* that are considered in the analyses are listed:

- AdaBoost, in package `ada` [CJM12]
- Conditional inference trees, in package `party` [HHZ06]
- Gradient boosting machine, in package `gbm` [Ro13]
- k-nearest neighbour classification, in package `kknn` [SH13]
- Support vector machine, in package `kernlab` [KSH+04]
- Linear discriminant analysis, in package `MASS` [VR02]
- Logistic regression, in package `stats` [VR02]. Binary classification decision derived using a probability cutpoint of 0.5.
- Least-squares support vector machine, in package `kernlab` [KSH+04]
- Naive Bayes, in package `e1071` [MDH+12]

- Multi-nominal regression, in package `nnet` [VR02]
- Random Forest, in package `randomForest` [LW02] using majority voting of classification trees
- Regularized discriminant analysis, in package `klaR` [WLL+05]
- Classification tree (CART), in package `rpart` [TAR13]

Dataset	Observations	Numeric	Integer	Factor
Blood Transfusion Service Center	748	2	2	0
ILPD (Indian Liver Patient Dataset)	579	5	4	1
Pima Indians Diabetes	768	8	0	0
Credit Approval	653	6	0	9
German Credit	1000	7	0	13
Spambase	4601	57	2	0
MAGIC Gamma Telescope	19020	10	0	0

Table 3.1: UCI classification tasks after pre-processing. The dataset ID, the number of observations and the number of features of the data sets are stored as numeric, integer or factor.

Even though most of the algorithms allow the adjustment of their parameters to increase the predictive performance, a fair comparison of the predictive performance would require deep knowledge of the algorithms or, alternatively, an automatic parameter tuning approach (as described in Chapter 6), which is not the focus of this chapter. Thus, for the analysis the algorithms are configured with either the most meaningful defaults, or, if available, the internal auto-tuning process was enabled.

The input datasets are listed in Table 3.1. Here, the most important criteria for the dataset selection are (a) being a 2-class classification problem, (b) a having sufficiently large number of observations to achieve accurate results in the profiling, and (c) having a realistic mixture of data types. As a pre-processing step, missing values in all observations were removed from the data.

The analyses in this chapter focus on the profiling of runtime and memory behavior of the algorithms, not on the performance of classification. However, the performance of the prediction is an important indicator that ensures an error-free program flow to avoid meaningless results. Therefore, the mean misclassification rate (as a performance measurement for the prediction) of a 10-fold cross validation is also monitored and presented in Table 3.2. Here, no errors that could result in constant or random predictions were observed.

	ada	ctree	gbm	kknn	ksvm	lda	logreg
IndianLiver	0.30	0.28	0.28	0.31	0.29	0.29	0.27
PimaIndiansDiabetes	0.24	0.25	0.35	0.27	0.24	0.23	0.23
GermanCredit	0.23	0.27	0.30	0.28	0.25	0.25	0.25
MAGICGammaTelescope	0.14	0.15	0.35	0.16	0.13	0.22	0.21
Spambase	0.05	0.09	0.39	0.08	0.07	0.11	0.07
BloodTransfusion	0.22	0.22	0.24	0.24	0.21	0.23	0.23
CreditApproval	0.13	0.14	0.45	0.16	0.14	0.13	0.15

	lssvm	naiveBayes	nnet	randomForest	rda	rpart
IndianLiver	0.29	0.45	0.29	0.29	0.31	0.34
PimaIndiansDiabetes	0.23	0.25	0.33	0.23	0.24	0.25
GermanCredit	0.26	0.25	0.30	0.23	0.29	0.27
MAGICGammaTelescope	0.20	0.27	0.24	0.12	0.21	0.18
Spambase	0.25	0.29	0.06	0.05	0.33	0.11
BloodTransfusion	0.26	0.25	0.24	0.25	0.24	0.21
CreditApproval	0.14	0.23	0.19	0.13	0.37	0.15

Table 3.2: Mean misclassification rates over 10-fold cross validation for all input data sets.

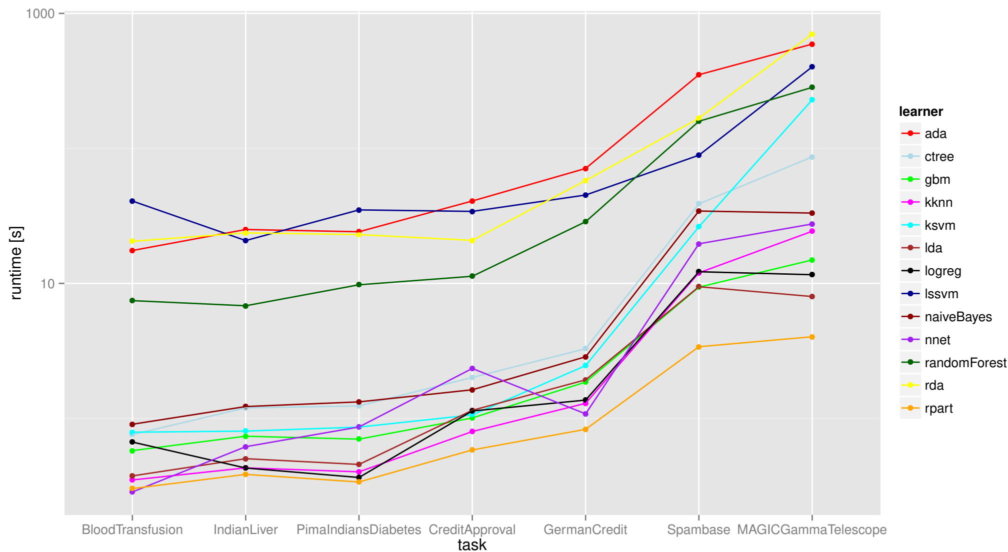

Figure 3.1: Execution times for 10-fold cross validation of all machine learning algorithms, including model fit, prediction and calculation of the misclassification error on all selected datasets. Y axis is on $\log_{10}$ scale.

To conveniently apply all machine learning algorithms on the receptive datasets, the machine learning R library `mlr` [BLK+16] is used. This library introduces some

overhead that influences both runtime and memory consumption. For example, it converts the input data from a matrix data type to a data frame or vice versa, which is a frequent pre-processing operation. However, it is expected that these operations do not affect the interpretation of the results since in a real-world scenario such transformations are done by the user.

Figure 3.1 gives an overview of the execution times of each algorithm applied to all datasets. To minimize the influence of start-up costs and provide a clear view on the bottlenecks, the following analyses will focus on those benchmarks that use the Magic Gamma Telescope dataset, as this input data set results in the highest aggregate runtime for all machine learning algorithms. In the next section, the results of the runtime behavior analysis will be presented.

3.3.2 Runtime Behavior Analysis

The runtime behavior analysis presents an overall runtime profile for each machine learning benchmark to expose runtime bottlenecks and suggest optimization ideas.

The following measurements are executed on a computer equipped with 2 AMD Opteron 2378 processors (quad-core, 2.4 GHz) and 16 GB of main memory, using a Debian Linux 7.3 as operating system. The profiling framework traceR is based on R version 3, compiled with the default compiler flags (just `-O2`) with GCC version 4.7.2. Under the default settings, the installed libraries (packages) included in the R interpreter are bytecode compiled and the default BLAS library (Basic Linear Algebra Subprograms) is used. The R bytecode compiler provides the option to compile R code into a bytecode representation. This enables optimizations which are generally beneficial for explicit loops in R code.

For the sake of clarity, the measurements for the runtime behavior analysis are summarized into *eleven categories* that can be split into *three groups* based on their optimization potential. Figure 3.2 illustrates the categorization for the runtime optimization potential of R Code.

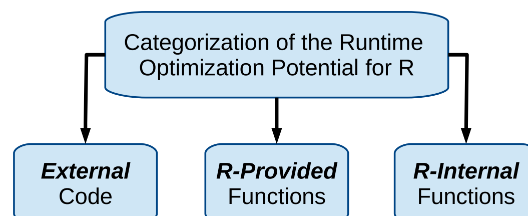

Figure 3.2: Categorization of the runtime optimization potential of R code.

The first group includes *External* code parts of the benchmarks like C or Fortran routines. R provides multiple ways for interfacing with external code to allow both the use of generic external libraries as well as libraries that are specifically written for use within R. The difference between those two is the way parameters are passed – for the generic interface, the R interpreter handles all required type conversions

itself. Libraries written specifically for R receive the internal representation of values instead, and handle any required conversions themselves. External code represents the lowest optimization potential as it is executed outside of the R interpreter.

The second group consists of functions directly provided by the R interpreter (*R-Provided*) like arithmetic operations or built-in functions. The categories that belong to this group will be later discussed in detail and are shown in Figure 3.5. Their optimization potential varies depending on the specific function.

The third group, where the highest optimization potential is to be expected, are the *R-Internal* tasks of the R interpreter that are not directly visible to an R programmer, but still important for the execution of an R program, such as memory management tasks. The categories that belong to the R-Internal tasks are represented in Figure 3.4 and will be explained in detail.

Figure 3.3 shows the percentage of runtime (see x-axis) spent in the eleven categories by the benchmarks (see y-axis). Since the execution time of the individual benchmarks varies greatly (see Figure 3.1), relative proportions of the runtime are used to ensure comparability. Here, a higher proportion of time spent in a category is considered to be a valid hint for the optimization potential of this category. For the development of optimizations that are beneficial for all machine learning algorithms, the analysis also focuses on the total time spent in a specific category for all benchmarks to approximate the potential for optimization.

However, as described above, the optimization potential can vary depending on the group a category belongs to (External, R-Provided or R-Internal). Furthermore, optimizations affecting one category may also influence the runtime spent in other categories. For example, an optimization that reduces the number of memory allocations would reduce the time needed for these allocations (*MemAlloc*) and also influence the time spent in garbage collection (*GC*) since fewer memory objects need to be checked. In the following, the results for the runtime spent in each group and category will be discussed.

External

The benchmark results in Figure 3.3 are sorted by the category *External*, which is the time spent executing non-R-code like C or Fortran libraries. This proportion varies between the benchmarks, as some of them are mostly implemented in R, while others make heavy use of external code. The highest amount of time spent on external code appears in the `nnet` and `randomForest` benchmarks, which both spend over 79.0% of their runtime outside of R. This demonstrates that the authors of these algorithms considered the runtime issues of R to be serious enough that they implemented large parts of their libraries in a faster language like C, even though this generally requires more effort than directly implementing an algorithm in R. On the other end of the scale, the `rda` and `lssvm` benchmarks both spend less than 0.1% of their runtime on external code. Coincidentally, they also are two of the slowest algorithms within the benchmark set.

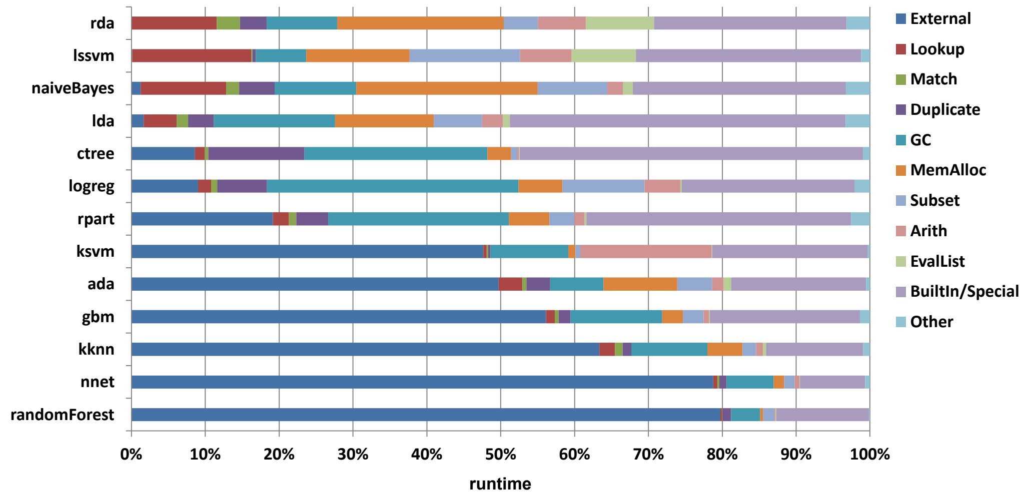

Figure 3.3: Runtime profiles relative to the total execution time of each benchmark.

However, one cannot conclude that the runtime of a benchmark is directly correlated with the use of R versus external code, as is emphasized by the runtime performance of the `ada` and `randomForest` benchmarks: In spite of them being among the four slowest algorithms, both spend more than 45.0% of their runtime in external code. Thus, even the use of external code does not guarantee fast execution times, as the complexity of the algorithm itself can largely influence its runtime and memory footprint. It is worth considering that the overall sum of time spent in external code parts for all benchmarks only accounts for about 15% of the total runtime. Hence, we can still expect significant improvements from optimizations on the R code side alone.

R-Internal

The group of the *R-Internal* functions is represented by six of the eleven categories – Lookup, Match, Duplicate, GC, MemAlloc and EvalList. These categories are also visualized in Figure 3.4. The gray arrows represent which of the R-Internal function categories can influence each other's runtime. For example, the more memory is allocated, the more time is needed in garbage collection (see arrow between GC and MemAlloc). As already mentioned, those categories have the highest optimization potential, especially for benchmarks that spend less time in external code like `rda`, `lssvm` or `naiveBayes`.

Lookup: For these, the amount of time spent in the category *Lookup* that looks up variables and functions during execution can be up to 16.1% of the total runtime. The reason for this large amount of lookup time lies in the behavior of the R programming language: before a function can be executed or a variable can be

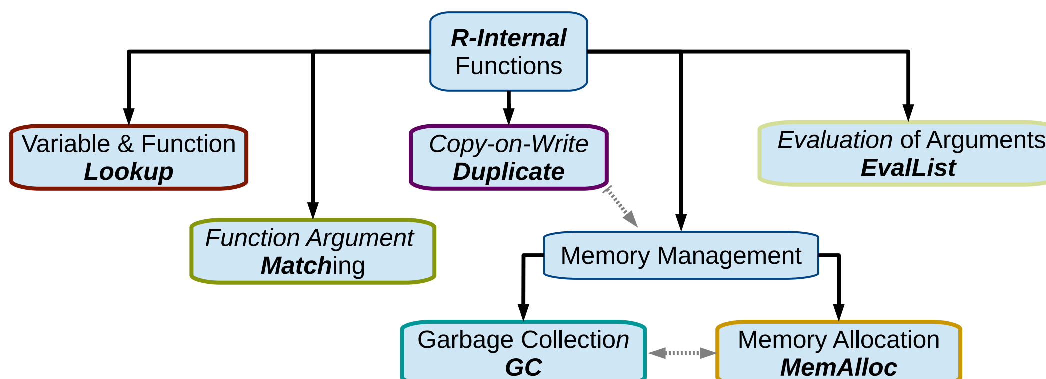

Figure 3.4: Runtime profile categories of the R-Internal functions.

accessed, the R interpreter has to look up its definition or value through a chain of environments. An environment provides a mapping from a symbolic name to a variable or function. Initially there is only the global environment, also known as the user workspace. Each function call adds one more environment to the end of this chain. The look up search has to be repeated every time an R program uses the name of a variable or function. This could be avoided by caching the result of the lookup function. Again, the R language creates additional problems for such an approach: R is a very dynamic language that allows a program to change the definition of a function during execution. This complicates a lookup cache as it would need to ensure that it never returns a cached lookup that has already been redefined by the R program.

Match: After the R interpreter has located a function definition that should be called, it needs to *Match* the arguments given in the call to the parameters given in the definition. Function arguments in R can be passed by name, by position or via the '...' argument, which is used to pass a variable number of parameters. The time spent on matching is up to 3.1% of the total runtime of the `rda` benchmark. Aggregated over all benchmarks, only 1.9% of the total runtime is needed for matching. Furthermore, the results show that there were no function calls with more than 17 parameters and over all benchmarks, 84.8% of all function calls had no or just one parameter. This indicates that the machine learning benchmarks rarely use the full flexibility of argument passing that R provides, which has a positive effect on runtime and is thus not a bottleneck compared to other R programs.

Duplicate: R uses a *copy-on-write* mechanism for function argument, the value of a parameter is in general only duplicated when it is modified by the called function, although exceptions exist. R uses this mechanism to implement *call-by-value* semantics. Duplication is marked as *Duplicate* in Figure 3.3. Its proportion varies between 0.3% (`ksvm`) and 13.0% (`ctree`) of the total runtime. Although

duplication itself contributes only 3.0% of the total runtime for all benchmarks, it increases the time spent in memory allocation (*MemAlloc*) and garbage collection (*GC*) because the duplicated values need to be stored and removed later. Besides the function lookup, the memory allocation and garbage collection are two main contributors to the runtime from the group of R-Internal tasks.

GC & MemAlloc: The *garbage collection* (*GC*) scans data that was allocated for values that are no longer in use and removes them. The time spent on garbage collection varies between 3.9% for `randomForest` and 34.0% for the `logreg` benchmark. Across all benchmarks, 9.0% of the total runtime is spent in *GC*. This runtime proportion is influenced by the number of memory allocations and by the memory footprint that is needed for the data structures of the R program.

The influence of the memory footprint can be illustrated by the `ksvm` benchmark, where 10.6% of the runtime is spent in garbage collection, even though only 0.9% of the runtime is spent in *memory allocation* (*MemAlloc*). Here, the time spent in memory allocation is low, because the benchmark allocates a small number of data structures (vectors), but the time spent in garbage collection is high, since each of those data structures has a large size and thus a large memory footprint (as described in Subsection 3.3.3). For other benchmarks the memory allocation can be a more important part of the runtime with a maximum of 24.6% in the `naiveBayes` benchmark. Here, a large amount of vectors with a small memory footprint is allocated. The dependence between memory allocation, memory footprint and garbage collection will be explained in more detail in Section 3.3.3 as this is one of the most important bottlenecks within the R-Internal tasks.

EvalList: The last category of the R-Internal tasks is *EvalList*, which represents a pre-processing step needed for a set of functions that are directly provided by the R interpreter. Thus, this category will be discussed after the categories of the *R-Provided* tasks.

R-Provided

The group of *R-Provided* functions includes three of the eleven categories – Subset, Arith and Builtin/Special (see Figure 3.3). Those categories are also illustrated in Figure 3.5.

Subset: For some of the machine learning benchmarks, a significant contributor to the overall runtime are subsetting operations (*Subset*) and the basic arithmetic operations (*Arith*). Those functions are part of the built-in functions of the R interpreter. The category *Builtin/Special* includes the time spent in built-in functions except for arithmetic and subset operations. Subsetting operations are used for the evaluation of *vector index expressions*. Those operations are, for example, used to generate training data subsets. The highest amount of runtime spent on subsetting

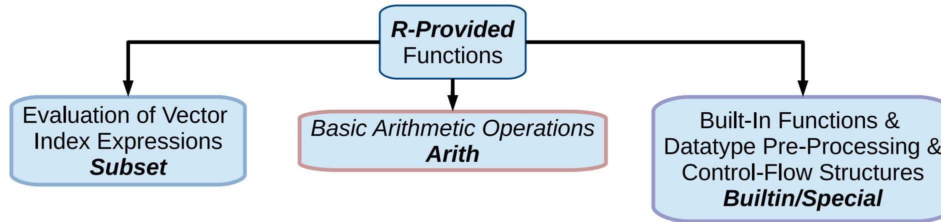

Figure 3.5: Runtime profile categories of the R-Provided functions.

occurs in the `lssvm` benchmark with 14.9%. Since subsetting has to allocate new memory to return results, it directly influences the time spent on memory allocation, as well as garbage collection.

Arith: The *basic arithmetic operations* (*Arith*) are operations like addition or matrix multiplication. Since R usually operates on vector data structures, these operations could benefit from the use of vector-oriented instructions in the CPU, which R currently utilizes only if special libraries are used. However, over all benchmarks, just 5.5% of the total runtime is needed for arithmetic operations. This implies that optimizing the runtime of these functions is unlikely to have a large impact on the overall runtime.

Builtin/Special: Before an arithmetic operation happens, the R interpreter has to run several pre-processing steps such as type checks or type conversions to ensure that the data has a valid format for the operation. This enables the dynamic type system of R. The runtime of those pre-processing steps is included in the category *Builtin/Special* and is part of the *built-in functions*. Those pre-processing steps are not only needed for arithmetic operations, but for almost all functions.

The overhead of these pre-processing steps could be reduced by the use of *function specialization*, which is a common compiler optimization. This optimization takes a generic function that can accept any data type and converts it into a specialized version that accepts only specific data types, which avoids the overhead needed for type checking. Such a specialization has been implemented at the byte code level in the Orbit VM [WWP14] in addition to other optimizations, yielding a total speedup of 3.5x over the standard bytecode interpreter on a set of R benchmarks that were mainly looping over data. The category *Builtin/Special* also contains the time spent on *special functions*. Special functions include R control-flow structures like `if` or `return`. Considering the time spent in this category over all benchmarks, 24.8% of the total runtime is spent in these functions. This number includes the time needed for type checks and data conversion, both of which could be optimized by the previously mentioned function specialization.

EvalList & Other: For the built-in functions provided by the R interpreter, another required pre-processing step is shown as *EvalList* in Figure 3.3. In this step, all arguments of a built-in function are evaluated before they are passed, which needs 6.3% of the total runtime over all benchmarks. R usually delays this evaluation when other types of functions like user functions or special functions are called. However, since the evaluation of arguments is needed to call built-in functions, its optimization potential is low. The final category called *Other* includes the remainder of the runtime like the start-up time of the interpreter and does not represent a viable optimization target.

Summed up, the runtime spent in memory allocation *MemAlloc* and garbage collection *GC* forms one of the bottlenecks with the highest optimization potential and is influenced by different categories like *Duplicate* or *Subset*. Thus, the next subsection analyzes the memory consumption of the considered benchmarks in more detail.

3.3.3 Memory Consumption Analysis

The analysis of the memory consumption has the goal to support the development of optimizations that reduce the footprint of the data structures used internally by the R execution environment (*Internal Data*) and data used by the algorithm itself (*User Data*). The memory allocation of those two groups of data structures and the runtime influence each other. Therefore, also the relationships between runtime and memory consumption behavior will be analyzed. In the following analysis, the same experimental setup is used as for the runtime behavior analysis of the previous subsection. First, the amount of allocated memory is compared to the amount of memory that was actually used during execution. Here, the amount of allocated memory ignores later removals of data by the garbage collector.

Figure 3.6 shows the maximum amount of memory that was used (*Memory Used*) compared to the total memory that was allocated (*Memory Allocated*) during execution for each benchmark. The memory used is measured as reported by the operating system using the `getrusage` system function. This function reports the maximum amount of memory (peak memory usage) that the program has requested from the operating system. The value is lower than the total allocated memory value since the garbage collection removes unused values from memory during runtime.

Across all benchmarks, 55.7 times more memory was allocated compared to the maximum amount used at once. This ratio can be used as one explanation for the amount of runtime spent on garbage collection and memory allocation. The benchmark with the highest ratio between allocated and used memory is `rda` with a factor of 512. This value also influences the 32.1% of runtime that it needs for memory allocation and garbage collection (see *MemAlloc* and *GC* in Figure 3.3). Furthermore, benchmarks with a high ratio between allocated and used memory and benchmarks that have in general a high memory consumption also suffer from long execution times. While `rda` and `ada` have the highest ratio, `lssvm` and `randomForest`

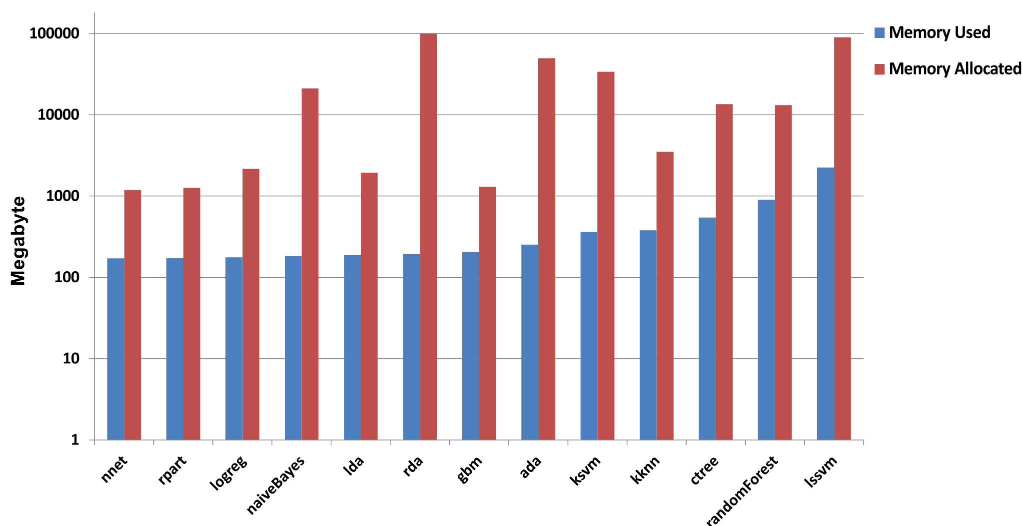

Figure 3.6: Maximum memory usage versus total memory allocation for each benchmark. Y-axis is on log scale.

consume the highest amount of memory. Those four benchmarks have also the longest execution times.

To provide a clearer view of the influence of the allocated memory on the overall runtime, the following analyses focus on the allocations of new data in memory, but ignore their later removal by the garbage collector. The first part gives an overview of the distribution of data structures that are allocated while the second part focuses specifically on vector data structures.

Data Structure Allocation

The data structures that are allocated during execution of an R program can be separated into two groups, visualized in Figure 3.7. Data structures that are primarily used for *R-internal* tasks are presented by the categories *Pairlists*, *Promises*, *Environments* and *Other*. Data structures that primarily hold *User Data* are included in the categories *External* and *Vectors*. Figure 3.8 shows the distribution of the allocated data structures. Since the memory allocations of the benchmarks vary, relative proportions are used to ensure better comparability.

Pairlists: *Pairlists* are mostly used as internal data in the R execution environment. Although pairlists are utilized in different parts within the R interpreter, one major contributor to their allocation is the creation of *argument lists* that are needed for each R function call. Thus most benchmarks with a high amount of R function calls suffer from a higher memory overhead induced by the allocation of pairlists, which negatively influences their runtime.

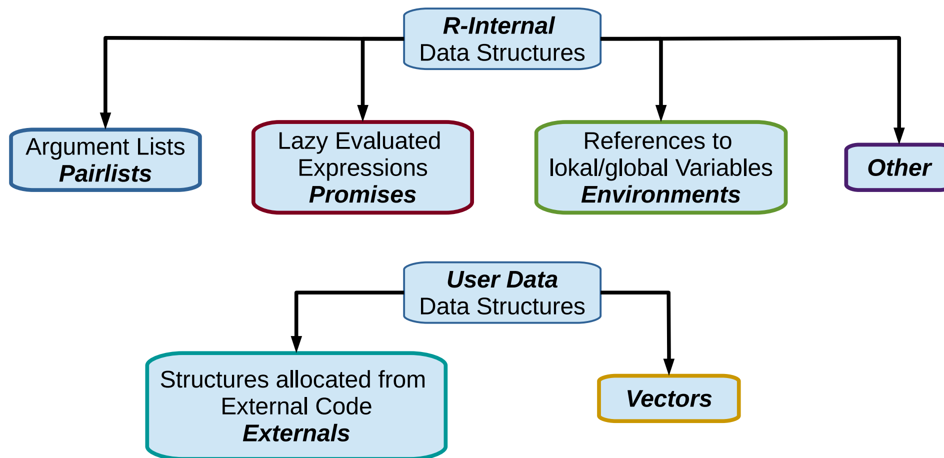

Figure 3.7: Memory profile categories of allocated R Data Structures.

The percentage of memory allocated for pairlists ranges from 1.1% for `ksvm` to 58.5%-60.4% for `rda` and `naiveBayes`. It is worth noting that a high amount of allocated pairlists corresponds to a larger proportion of time spent in the runtime category *Lookup* shown in Figure 3.3 (e.g., in `rda`). The overhead incurred by pairlist allocation triggered from R function calls could be reduced by a dynamic compilation approach, which could reduce the number of function calls by *inlining* small functions into their caller.

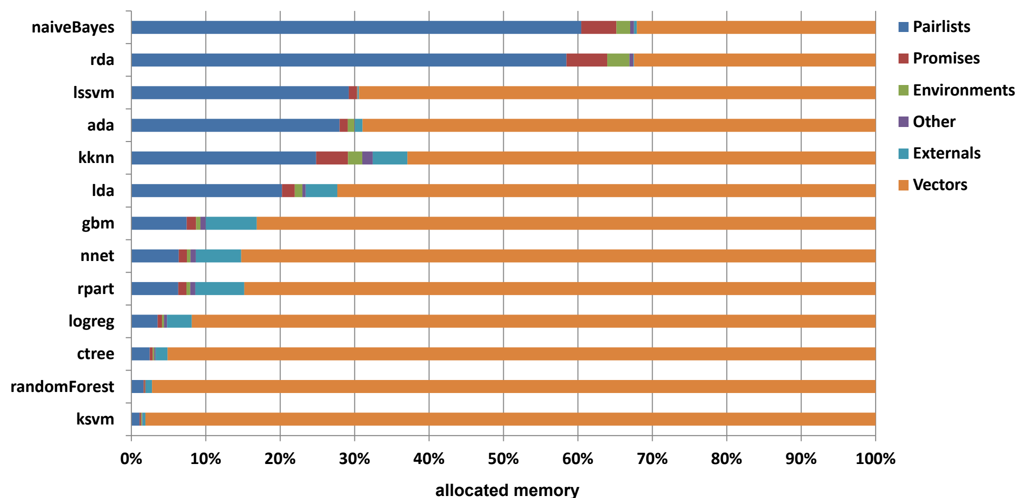

Figure 3.8: Memory profiles relative to the total memory allocation of each benchmark.

Promises: The next category discussed are *Promises*. Due to the functional characteristics of the R language, a function argument can not only be bound to a simple value, but also to a complex expression like a function call. Such an expression is only evaluated when its result is really needed, which can happen either in the immediate callee itself, or when the callee passes a not-yet-evaluated expression to another function that then requires its result. This mechanism is called *lazy evaluation*, and is implemented by boxing each function argument in a so-called "promise".

A promise needs 56 bytes (on a 64 bit system), besides additional information stored in the header, each *Promise* contains a reference to the original argument and its corresponding environment. Aggregated over all benchmarks, only 2.5% of the total allocated memory is used for promises, even though the absolute values reveal some optimization potential. Three of the benchmarks (`lssvm`, `naiveBayes` and `rda`) use more than 1 GB of memory allocations just for promises with a maximum of almost 6 GB for `rda`. The analysis with traceR also shows that in 86.4% of all cases, the creation of a promise was not necessary since its evaluation happened directly in the function that it was initially created for. However, as explained by Morandat et al. [MHO+12], it is not always possible to replace the creation of promises with eager evaluation.

Environments: The last important category in the group of interpreter-internal data structures are *Environments*. Similar to promises, they only contain references to other values and are mainly created when an R function is called. Considering all benchmarks, 1.2% of total memory allocations is used for environments. Since the interpreter needs to search through the chain of environments during lookups, this apparently small value has a strong influence on the runtime required for lookups. As described in subsection 3.3.2, lookups are the second-largest bottleneck in the interpreter internal tasks.

There are also a few additional internal data structures that require memory allocation, but since they use less than one percent of the total allocated memory of all benchmarks, the *Other* category is not an interesting optimization target.

Summed up, all internal data structures described above contribute 40% to the total allocated memory. Thus, almost half of the allocated memory is used for the execution of the R program, and not for the user data it processes.

User Data: The user data structure allocations are divided into allocations that are triggered by external code parts (*Externals*) and the allocation of *Vectors*. Memory allocation from external libraries requires just 0.5% of the total allocated memory, while vectors account for 61.2% of the total allocated memory over all benchmarks. As such, *Vectors* are the biggest consumer of memory allocations in the benchmark set and will therefore now be analyzed in more detail.

Vector Data Structure Allocation

Vectors are the most important data structure in the R language, and higher-dimensional structures like matrices and arrays are internally constructed from them. When R allocates memory for a vector, it differentiates between different classes of vectors to reduce the memory allocation overhead. Depending on the size of the vector, the memory allocations are pooled for small vectors or allocated via the *malloc* C library function for large vectors. Small vectors can store up to 16 double or 32 integer or logical values (up to 128 bytes), while large vectors are used when the total size of elements exceeds this limit. In addition to these two categories of vectors, the analysis also considers vectors with exactly one element separately from the small vectors (as well as vectors with zero elements), since single-element vectors have a special optimization potential:

In R, each vector needs a header block so it can be integrated in the memory management subsystem of R. The size of this header is 40 bytes (on a 64 bit system) per vector. For vectors with fewer elements, the relative overhead of this header increases. The worst case are single-element vectors: A 40 byte header is needed to manage an object whose size is just 4 or 8 bytes, thus in the worst case the header needs 10 times more memory compared to the real data. This suggests an optimization potential for introducing *scalar values* that are not boxed within a vector and thus could use a smaller header. If such a scalar value is only used within a function, a just-in-time compiler may even be able to keep the value in a CPU register instead of storing it in main memory, saving the time for both allocation and garbage collection. However, the potential of reducing the memory footprint of a benchmark by introducing scalar values is only high if those single-element vectors dominate.

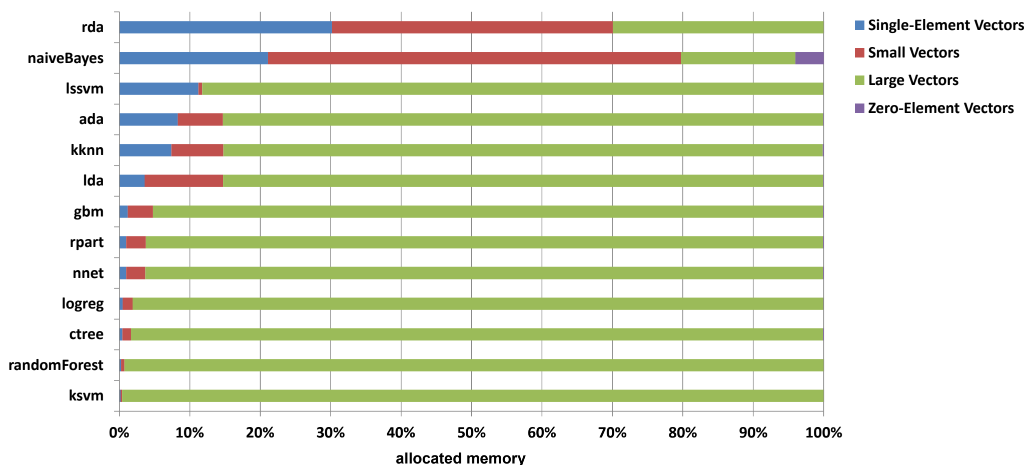

Figure 3.9: Vector memory profile relative to the total memory allocation of vectors for each benchmark.

Figure 3.9 shows the proportions of the allocated vector sizes in relation to the total vector memory allocation for each benchmark. In most benchmarks the proportion of allocated memory for *Large Vectors* dominate. Even though Figure 3.9 shows only the ratios for the processing of the Magic Gamma Telescope data set, the distribution between *Large* and *Small/Single-Element Vectors* is roughly the same for all input data sets. `NaiveBayes` is the only benchmark where *Zero-Element Vectors* appear, reaching about 4.0% of the total vector memory allocated compared to less than 0.2% for all other benchmarks. The benchmarks `rda` and `naiveBayes` use a much higher percentage of single-element vectors compared to the other benchmarks. This factor also influences their time spent on *MemAlloc* (see Figure 3.3) along with a large number of pairlists created in these benchmarks.

However, over all benchmarks *large vectors dominate* and only 20.4% of all allocated vector memory is used for single-element or small vectors. Thus, for most benchmarks, optimizing the allocation of large vectors has a high optimization potential compared to optimizing the allocation of single-element vectors by introducing scalar values. The size of the underlying memory region of a vector is fixed at allocation time and cannot be changed; changing the length of a vector hence prompts the R interpreter to create a copy with the new length. The old instance might be discarded by the garbage collector if no more references to it exist. This produces a huge copy overhead, especially for large vectors that can potentially span multiple memory pages. Here, a general memory optimization like sharing of memory pages could be used to reduce the memory footprint and thus the runtime needed for memory allocation and garbage collection.

3.4 Summary

This chapter presented an analysis of the most popular statistical machine learning R algorithms. With this analysis, detailed insights into the runtime and memory issues of these algorithms were gained, with the goal to outline optimization approaches that overcome the discovered bottlenecks. The results can guide the development of alternative R execution environments and support optimizations for the original R implementation that enable the algorithms to scale to larger problem sizes. For the analysis, an R profiling framework called traceR was redesigned and enhanced. All algorithms were applied to real-world data sets from the UCI [BL18] repository.

For the algorithms that are implemented mostly in R code, the runtime overhead incurred by function calls (*Lookup*) that triggers the allocation of argument lists (*Pairlists*) was one significant contributor to their runtime. A dynamic compilation approach may be able to provide improvements for this bottleneck, as it could reduce the number of function calls by inlining small functions into their caller. Another area where such an approach can be helpful is the time spent on functions provided by the R interpreter (*Builtin/Special*): to support the dynamic type system of the R language, these functions must perform type checking and conversion of

their arguments. Using specialization techniques, this overhead could be avoided whenever the data types used in the call are known to the compiler.

Overall, the analysis demonstrates that the memory management (*GC* and *MemAlloc*) is a major contributor to the total runtime of the benchmarks and has a high optimization potential. The time spent on memory management is influenced by different R-internal and R-provided functions (like *Subset* and *Duplicate*), as well as by the number of data structures that are allocated and their footprint. Vector data structures on the user data side and pairlists on the side of the data that is internally used are the main data structures that are used in memory allocation. Here, especially large vectors that can span multiple pages of memory dominate.

The analysis suggests a general memory optimization like sharing of memory pages to reduce the memory footprint and thus the runtime needed for memory allocation and garbage collection. Such a memory optimization can be done without the need of implementing a new compiler or significant changes of the R execution environment, which is important for compatibility reasons (keep in mind that R has no language specification). The next chapter will therefore present an optimization of the memory consumption for R and thus for the analyzed machine learning algorithms.

Chapter 4

Efficient Memory Utilization for Machine Learning Algorithms

This chapter presents an optimization approach for efficient memory utilization of machine learning algorithms that are written in the R programming language. It is based on the work of Kotthaus et al. [KKE+14]. Optimizations for an efficient resource utilization of statistical learning algorithms based on R can only be profitable if they do not break compatibility with the software libraries most R programs are based on and simultaneously address the real resource bottlenecks of those algorithms. As shown by Morandat et al. [MHO+12], as well as by the analysis of bottlenecks in machine learning R algorithms presented in the previous Chapter 3, the R execution environment has several drawbacks leading to slow and memory-inefficient program execution. Especially in the domain of machine learning algorithms, the R interpreter induces a large memory overhead due to wasteful memory allocation policies [KKL+14].

In R programs, most data structures are vectors (or based on vectors). Depending on the size of these vectors, the R interpreter chooses between multiple allocation strategies to reduce fragmentation. When the size of a vector is above a certain threshold, the interpreter allocates a large vector. As shown in the memory consumption analysis presented in Section 3.3.3, the allocation of large vectors dominates in most of the analyzed machine learning algorithms. For each large vector, a dedicated block of memory is allocated, potentially spanning multiple pages. These pages, even when unused, take up memory. When the amount of memory required for computations exceeds the physical memory available to the application, the execution is drastically slowed down by frequent page swaps that require disk I/O, a phenomenon also known as “thrashing”. The performance penalty due to thrashing might render complex computations entirely infeasible.

Existing R optimizations are mostly based on dynamic compilation or native libraries. However, the performance penalty incurred by thrashing is several orders of magnitude higher than the benefit either of both methods can provide. The basic features of R are extended by over 11,000 packages available in public repositories like CRAN [R C18b]. There also exist dedicated libraries for processing large datasets, particularly addressing more efficient memory management strategies (e.g., using sparse matrices [KN17]). However, libraries are often tailored to concrete applications, and thus cannot simply be used by arbitrary R programs. In contrast, the optimization approach in this thesis targets the memory management layer

between the R interpreter and the operating system, making it applicable for any R program.

The goal of the memory optimization approach presented in this chapter is to enable efficient memory utilization especially for memory-hungry R applications like machine learning algorithms. It reduces the memory overhead for large data structures by ensuring that memory will only be allocated for memory pages that are definitely required by the R algorithm (avoiding duplications) and by enabling the sharing of already allocated pages (deduplication).

The sharing of memory contents usually incurs a time-memory-trade-off: the more aggressively pages are shared, the more time must be spent to unshare them when they are modified. Blindly optimizing would therefore incur a larger runtime penalty compared to a strategy that takes into account the expected use of the memory, avoiding sharing when it is expected that no memory could be saved. While there already exist similar OS level optimizations such as lazy page loading (meaning a page is only allocated when it is written to or read from for the first time) or sharing of pages with the same content (deduplication), these optimizations lack knowledge about the specific memory behavior of the runtime environment. This reduces the ability of the operating system to make qualified decisions about optimizing the memory usage. For example, the OS cannot know if a memory block requested by a function will be written to immediately or only at a later time. As a remedy for this, the approach presented in this thesis uses additional information from the R runtime environment to guide the optimization, taking into account, e.g., the short-term usage pattern of a memory block. Its evaluation is based on different benchmark sets, including benchmarks that were used to evaluate related approaches, as well as statistical machine learning algorithms. Especially when the memory consumption hits the point that the OS starts to swap out memory, the new optimization approach is able to speed up computation by several orders of magnitude.

This chapter is structured as follows: Section 4.1 gives a survey of related memory optimization approaches and general optimization approaches for the R language. The fundamentals of R's memory management are explained in Section 4.2. Section 4.3 presents the optimization strategies for efficient memory utilization of R algorithms developed in this thesis. The evaluation of the memory optimization is presented in Subsection 4.3.5. Finally, Section 4.4 concludes with a summary.

4.1 R Optimizations and Memory Footprint Reduction - Existing Approaches

A number of projects already work on various optimizations for the R language environment. Some of these projects like FastR [KMM+14; SWH+16], Renjin [Ber18] or Riposte [TDH12; TDH14], reimplement the original interpreter in another language such as Java or C++. These approaches benefit from optimizations available in their

runtime environments. However, they cannot yet guarantee full compatibility with existing R programs and libraries due to the complex and evolutionary development of the R language and its missing formal specification. Optimizations from other projects like pqR [Nea18] or Orbit VM [WWP14; WPW15] concentrate on specific bottlenecks of the R interpreter by using, e.g., function specialization or by introducing scalar data optimizations. For other interpreter-based dynamic languages, there are similar approaches realized in projects like MaJIC [AP01] and PyPy [BCF+09].

In contrast to the above-described approaches, the memory optimization of this thesis works on a layer between the R interpreter and the OS. This enables optimizations of arbitrary R applications with only small modifications to the R interpreter and without requiring application changes. Thus, in the following, the related system-level approaches for reducing memory utilization will be discussed.

In general, related work on utilizing main memory more efficiently can be categorized into two groups: memory compression approaches, often influenced by embedded systems resource constraints, and memory deduplication, which is mostly used in virtualization. Memory compression tries to reduce the swapping activity of a system by compressing memory contents instead of swapping pages to the secondary storage. A memory trace-based evaluation of different deduplication and compression approaches is presented by Deng et al. [DSH13], showing that deduplication yields better results than memory compression. Compression approaches share the drawback that a significant amount of processor time is spent on compressing and decompressing memory contents.

In contrast, *memory deduplication* reduces the memory overhead by mapping virtual pages with identical contents to a single physical page. This is often beneficial in virtualized environments where large amounts of read-only memory, such as shared libraries, are used in multiple virtual machines [SK12]. However, deduplication can introduce significant computational overhead, since the contents of pages have to be scanned periodically in order to identify pages with identical content. An often used implementation of deduplication that has been the subject of multiple improvements is available in Linux as the *Kernel Samepage Merging* (KSM) [AEW09]. For example, Miller et al. [MFR+13] use I/O-based hints to prioritize the scanning of pages that were recently read from disk, reducing the time needed to detect shareable pages. KSM has also been optimized in [CWC+14] by introducing a classification scheme based on access characteristics, comparing only pages within the same class to reduce the overhead of page scanning.

However, all of these improvements are still reactive, meaning that they can only eliminate duplicate pages after they have been created, not prevent them. Although some information is used to improve the time needed to detect duplicates, the application-specific knowledge of why the data was copied in the first place is ignored.

Sharing memory pages within a single process appears to be a rarely-used concept: on Linux, it is automatically used to map a set of newly allocated virtual pages to a single physical page filled with null bytes. This can cause performance

issues in high-performance environments since each write to any newly allocated page will trigger a page fault. Here, an enhancement by Valat et al. [VPJ13] was proposed that avoids unnecessary page removal when the application knows that it will overwrite a page in the near future. A language-level version of this *copy-on-write* technique, operating on objects instead of memory pages, is sometimes implemented using reference counters. Tozawa et al. have analyzed the details of PHP's copy-on-write scheme and proposed improvements in [TTO+09]. The R language also implements a copy-on-write scheme. Here, the complete object (potentially spanning multiple pages) is copied when it is modified, resulting in page duplications for partial modifications.

Compared to the above-described deduplication approaches, the memory optimization in this thesis employs specific knowledge about the interpreter state to reduce the number of pages that need to be scanned for identical content. Here, scanning itself has a low overhead, since it is content-based and only checks for all-zero pages, which can terminate at the first non-zero element. In contrast to the previously described, purely reactive approaches, the memory optimization described in this chapter *proactively* avoids the main sources of identical-content pages from object allocation and duplication by optimizing the copy-on-write mechanism for partial object modification. It is furthermore aware of data structures used by the R interpreter and their contents. Thus, a better prediction and avoidance of unnecessary deduplication is enabled.

The next section gives an overview of the memory allocation behavior of the R runtime environment.

4.2 Memory Allocation in the R Runtime Environment - Fundamentals

The lifecycle of an object, (e.g., a vector data structure) in the R execution environment starts with its allocation. In R vectors are assumed to be made up of a contiguous block of (virtual) memory. Depending on the size of the object, the R interpreter uses a system of multiple memory pools for vector objects with a data size of up to 128 bytes. For larger vectors, memory is allocated directly via the `malloc` C library function instead of pooling the allocations. This reduces the memory fragmentation when many small objects are created and some of them are released. The R language does not require the programmer to explicitly manage memory, thus a garbage collection is needed to automatically free memory. The garbage collector in R is a mark-and-sweep, non-moving, generational collector. It can be manually triggered, but it also runs automatically when the interpreter is in danger of running out of heap space.

The R interpreter ensures that an allocated object is always initialized – either by explicit initialization or implicit by writing the results of a computation to it. After the object is allocated and initialized, it can be used as input for various R

functions such as the plus operator. The fact that functions can modify an object, in conjunction with R implementing *call-by-value* semantics, means that objects need to be copied when being passed to a function. However, at this point a copy-on-write optimization is triggered: copying an object is done by merely sharing the reference; the actual copy is delayed until the object is modified (if at all). The interpreter now has two references to the same object that may be modified later. When this modification happens, the copy process is triggered and a full copy of the affected object, potentially spanning multiple pages, is created using the interpreter-internal *duplicate* function. This is illustrated in Figure 4.1.

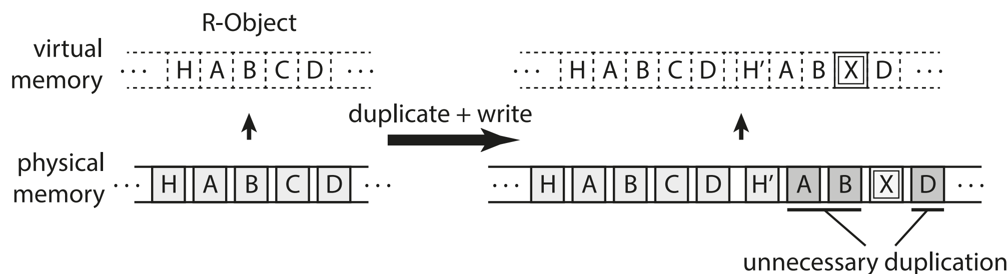

Figure 4.1: Example of the copy-on-write mechanism in the GNU R interpreter. R copies (duplicates) at object level instead of page level granularity.

On the left-hand side, a large R vector object consisting of a header *H* and four pages *A* to *D* is shown both in *virtual memory* on the top (marked with dotted lines) and its corresponding allocated *physical memory* on the bottom (solid lines). On the right hand side, the situation after a duplication that was triggered by a write access is shown. Now there are two R objects shown in the virtual memory on top and their corresponding physical memory on the bottom. In one of the copies, page *C* was modified and is now marked as *X*, the copy has its own header *H'*. Although unmodified, the R interpreter needs to use additional memory to create duplicates of pages *A*, *B* and *D* (marked in gray) since it assumes that objects are organized as contiguous blocks of memory and thus it has to duplicate at *object level granularity*.

The memory optimization presented in this chapter has the goal to reduce this memory overhead by copying only parts of the object instead, sharing the same memory pages between multiple objects as long as they are not modified. This scheme is transparent to the interpreter's memory management including the garbage collection, requiring only small changes in memory allocation and freeing, as well as in the duplicate function. The details of this optimization will be presented in the next section.

4.3 Memory Footprint Reduction via Dynamic Page Sharing Strategies

This section describes the optimization strategies for efficient memory utilization of R algorithms developed in this thesis. The approach that proactively avoids the duplication of memory pages for R is described in Subsection 4.3.1. It is based on optimizing the allocation and duplication mechanisms of the R interpreter. This approach is further refined by using static annotations that reduce the optimization overhead, presented in Subsection 4.3.2 and by a dynamic refinement using a page content analysis for page deduplication to increase the amount of shared memory, introduced in Subsection 4.3.3. Subsection 4.3.4 presents the interplay of the described memory optimization strategies.

4.3.1 Page Duplication Avoidance

As shown in the previous section, the R interpreter can only allocate complete objects that potentially span multiple pages. The first part of the memory optimizations of this thesis is based on the object allocation mechanism of R. To enable the allocation and thus the sharing of memory at *page level granularity* instead of object granularity, a custom memory allocator is employed when a large vector has to be allocated, as shown in Figure 4.2. When the internal function of the R interpreter *allocVector* is called to allocate a large vector, the optimized interpreter selects between the *custom allocator* to share memory on page granularity or the *default malloc* function if this is not required. In both cases, the allocated memory is accessible within the address space of the R interpreter.

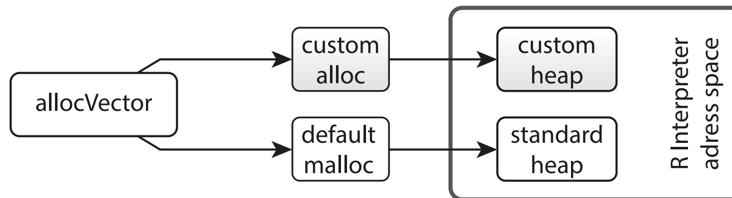

Figure 4.2: Memory allocation scheme for dynamic page sharing strategies.

The custom allocator uses a memory management scheme similar to standard virtual memory schemes commonly used in OS kernels. For ease of implementation, it is completely implemented in the user space.

The downside of such a user space implementation is that it needs to replicate certain data structures that are already present in the OS (e.g., for mapping virtual to physical memory) because those kernel data structures are not sufficiently exposed to user space. This replication could be avoided by implementing the optimization in the kernel (cf. [KKM16]), which however is significantly more invasive and not applicable in a lot of environments where the user has no control over the kernel

running on the machine. Considering that the user space has no direct access to physical memory, a single file located on a RAM disk (see *custom heap* in Figure 4.2) is used.

The allocation of physical memory from this file is realized via a simple free-bitmap based allocator. The file can be dynamically enlarged if needed. Mapping physical pages into the virtual address space of the R interpreter can be accomplished by utilizing the `mmap` Unix system call. For changing the access permissions of these physical pages the `mprotect` system call that modifies the settings of the memory management unit of the processor is employed. Besides these system calls an additional page table is needed to enable the mapping from a virtual address to a physical address. For simplicity reasons a hierarchical page table with the same four-level structure as used by the processor is implemented. To enable the sharing of pages, the user space memory management system needs to map the same physical page to multiple locations in virtual memory. Therefore, a reference counter is required for each physical page. A reference counter greater than 1 indicates that the page is shared between multiple objects or multiple times within one object.

To avoid the zero-initialization of allocated large vector objects a *global shared zeroed page* is utilized. This also ensures that memory is only allocated for pages that are actually written to at a later time. Figure 4.3 illustrates an example for this optimized R object allocation. Here, the custom memory allocator was asked to allocate an object with a total size of five pages.

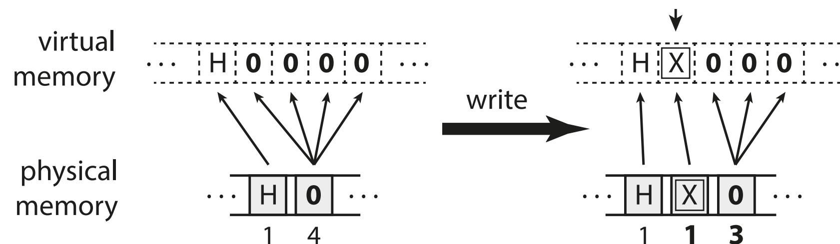

Figure 4.3: Optimized object allocation via sharing a global zeroed page.

While the object has the requested size of five pages in virtual memory (dotted, left upper part), physically it only consists of two pages (left lower part). Those two pages are a single non-shared page, in the beginning, marked with *H* for header, followed by a shared page, marked with ***0***, called the global zeroed page. The numbers in small print below the physical pages are the *reference counters*. The zeroed page has a reference counter of 4 since it is shared four times within the allocated object (mapped four times into virtual memory). The shared zeroed page is filled with zero-bytes. The concept of prepared zeroed pages is already implemented in OS kernels. However, the standard R interpreter does not benefit from this concept since it immediately writes to all memory that it allocates for initialization. The non-shared initial page *H* is required as it will contain not just data but also

the object header. The R interpreter writes this object header to the front of the allocation area. Since it will be updated frequently (e.g., during garbage collection), it is not shared between multiple objects. Since the header page H is mapped only once, its reference count is 1.

The R interpreter now has the illusion that it has allocated five pages of memory, even though only two pages are allocated physically. To sustain this illusion, the optimized allocation mechanism has to ensure that any write access to a virtual page that points to a shared physical page can be detected and handled. If such a write access is not handled correctly, it would affect not only the intended virtual page but also all virtual addresses where the same physical page is shared. For the example in Figure 4.3 this would mean that a write to one of the four virtual instances of the zeroed page would be mirrored in its other three instances, resulting in incorrect object contents. Therefore, all pages with a reference counter greater than 1 are marked as *read-only*, ensuring that a write access triggers a segmentation fault. This fault is caught by a *signal handler* that performs the unsharing of the affected page. To determine the affected physical page the handler uses the virtual address of the write access. It then allocates a new page, copies the contents of the original page to it and replaces the page that caused the segmentation fault with the new one. The resulting situation is shown on the right side of Figure 4.3: One of the instances of the zeroed page which was written to was replaced with a new page marked with X. This updates the reference count of both the zeroed page and the newly allocated page. Since the new page is only mapped once, it can now be marked read-write and further accesses would not require special handling anymore.

As noted in Section 4.2, the R interpreter can only copy on the object level. Thus, if an object consists of multiple pages, parts of the copy may end up with identical content as the original (see Figure 4.1). To avoid this, the *duplicate mechanism* of the interpreter is optimized by improving the granularity of the copy from object level to page level. While the allocation optimization avoids the immediate allocation of pages by using the global zeroed page, the duplicate optimization allows the reuse of already-allocated pages of the original object instead of allocating new pages. An example of the duplicate optimization is shown in Figure 4.4.

On the left side the situation before the duplication is shown: An object occupies five virtual pages, two of which reference the global zeroed page. Unlike the original R interpreter that would need to allocate five new pages for the copy of this object, the optimized version reduces this to a single allocated physical page. This is shown on the right side with the original object on the top and its copy on the bottom. Here, a *virtual-only copy* of the first page that contains the object header is not created, since the header of the copy is updated immediately by the R interpreter after the duplication which would trigger an unsharing of this page. To avoid the overhead of this event, the optimized duplication immediately creates a physical copy of the header page. Since most of the pages of the original object are now mapped twice in virtual memory, the reference counters of the corresponding pages

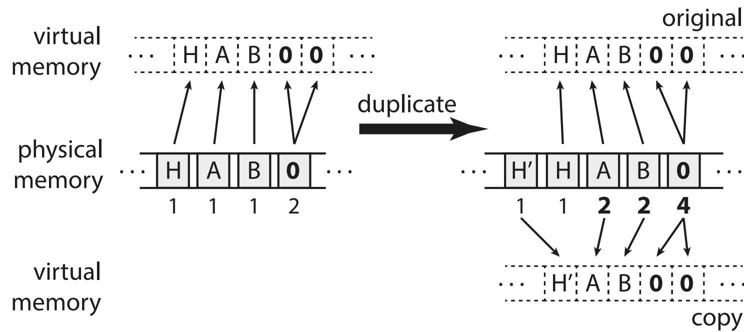

Figure 4.4: Optimized copy mechanism on page-level instead of object level granularity via page sharing.

are updated and both the original and copy are marked as read-only to allow for unsharing on write accesses.

Overall, the finer copy granularity of the optimization enables storing both the original and copied objects from the example in just five pages of memory. In contrast, the original R interpreter would need ten pages of memory to store the same objects. Although the mechanism of sharing pages during allocation and duplication described above always result in a valid view on memory for the interpreter, there are cases where additional overhead is caused that can be avoided by further refinements described in the next subsection.

4.3.2 Static Refinement via Annotations

To reduce the runtime overhead caused by proactively avoiding page duplications, a static refinement that consists of two kinds of annotations is applied. The first annotation is based on the expected utilization of an object immediately after allocation and the second annotation is based on the size of the allocated object.

The optimized memory allocation described in the previous subsection (see Figure 4.3) reduces the memory footprint by utilizing a global zeroed page, assuming that not all pages of the allocated object will be written to immediately. However, this assumption is not always valid, as for example (built-in) vector arithmetic functions in the R interpreter allocate a new object and immediately write to all pages of it to store their results. This would cause a segmentation fault for the first write of every page, triggering the memory allocation for all pages of the object. These segmentation faults cause runtime overhead that would not occur when allocating an object with non-shared pages.

To avoid this overhead, *annotations* are placed in the C source code of the R interpreter built-in functions where newly allocated memory is completely overwritten directly after allocation. Here, the custom allocator returns an object where every virtual page references a new physical page, so no segmentation faults will be triggered by write accesses. Although these R objects do not save memory on

allocation, they still have the opportunity for later optimizations, e.g., when they are duplicated. Currently, the annotations for these *"full-overwrite"* functions need to be manually in the R interpreter's C source code by locating calls to *allocVector*, followed by loop structures that write to every element of the newly-allocated object. Those manually placed annotations could also be automated by a static code analysis that checks for allocation calls followed by loops that write to the newly-allocated object similar to the pseudo-code in Figure 4.5.

```
1 object = allocVector(length)
2 for (i = 0; i < length; i++):
3     object[i] = calculation(i)
```

Figure 4.5: Pseudo-code of a "full-overwrite" - where pages are not shared during allocation.

The second annotation for reducing the runtime overhead incurred by the optimization relates to the size of the object that is allocated. The R interpreter can allocate objects with a variety of sizes, not all of which span multiple pages. The optimized custom allocator is therefore only enabled for object sizes that indicate a potential for page sharing. Here, the potential is limited for smaller objects. The first page of an object stores not just data but also the frequently modified object header that is therefore never shared. Thus R objects smaller than two pages of memory are passed to the standard, non sharing memory allocator. This size limit could also be used as a tunable parameter to select a trade-off between memory savings and runtime overhead.

In addition to the above-described static refinements, an additional dynamic refinement for increasing the number of shared pages is applied and will be presented in the next subsection.

4.3.3 Dynamic Refinement via Page Contents

During the execution of an R program, allocated objects are updated with the results of calculations. Those updates can result in multiple distinct pages with same contents, which opens up the opportunity for sharing those pages. The general idea of locating identical objects in a system and saving memory by reducing them to a single object is known as deduplication (as described in Section 4.1).

The memory optimization of this thesis employs a restricted version of locating identical contents. Here, the content scan only checks for pages identical to the already existing global zeroed page. The *deduplication of zeroed pages* is illustrated in Figure 4.6. On the left side, the situation before the page content scan is shown where an object contains two identical zero pages. One of those pages is already mapped to the global zeroed page (shown in bold), while the other uses a separate physical page. On the right side, the situation after deduplication is shown. Here,

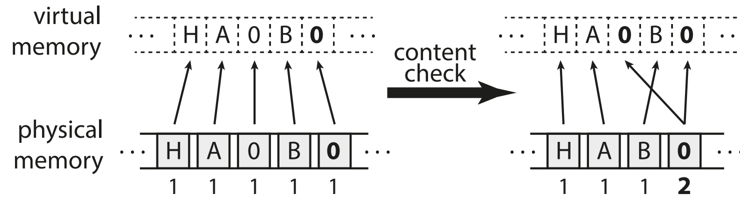

Figure 4.6: Deduplication optimization for zeroed pages.

the *content check* has detected the separate copy and mapped its virtual page to the global zeroed page, freeing the memory used for the unnecessary duplicate.

Although a general scan that is able to detect duplicated pages with arbitrary content, could be applied, such a scan would incur a significant runtime overhead (e.g., due to the calculation of hash values) compared to scanning just for zeroed pages. While a scan for zeroed pages can use an early abort condition as soon as a non-zero element is found, a scan for arbitrary content would need to check the full content of all pages in the system. The overhead incurred by deduplication of zeroed pages is influenced by the frequency of the content check and by the number of pages that need to be scanned. To reduce this overhead, the scan is only activated after the completion of a garbage collection in the interpreter. This avoids scanning of pages that would soon be discarded and also provides a natural regulation mechanism for the frequency of content checks as the frequency of garbage collection depends on the memory requirements of the executed program.

With the above-described deduplication optimization, pages that were previously excluded from sharing the global zeroed page, e.g., in arithmetic vector operations (as described in Subsection 4.3.2) can now be dynamically shared. Thus, both the static and the dynamic refinements of the memory optimization complement each other. The interaction of the refinement strategies and the general page duplication avoidance strategy will be described in the next section.

4.3.4 Dynamic Page Sharing Optimization

The main functionality of the optimization is shown as pseudo-code in Figure 4.7. The optimized custom allocator function is presented in lines 1 to 22. It includes both static refinements (as described in Subsection 4.3.2). The first refinement (line 3) uses the size of the allocated object to decide if the optimization should be enabled. If the object is sufficiently large, the allocator first needs to allocate a region of virtual memory to map the object (*alloc_virtspace*, line 8). This is emulated with the *mmap* system call, that lets the kernel select a free region of virtual memory. Although the *mmap* call maps data into this region, this mapping can be later overwritten with the *map_page* function call (line 21).

The second static refinement is realized via the *expect_full_write* parameter. Callers that immediately overwrite the object after allocation set this parameter to

indicate that they will not benefit from an initialization via the shared zero page. This parameter is used in line 14 to select if the allocator should return references to the global zeroed page or directly allocate physical memory. Line 21 then calls the *map_page* function to map the selected page for the new object.

The pseudo-code of the *map_page* function is shown in Figure 4.8 (lines 1–12). It receives two parameters, a physical page and an address of a virtual page to which the physical page should be mapped. First, it adds a mapping from the virtual page to the physical page to the page table. Then the actual mapping in virtual memory is updated in line 3, which uses the *remap_file_pages* Linux system call to change an existing mapping. Finally, the reference counter of the physical page is incremented (line 6). If the reference counter is greater than one (see line 9), the page is marked as read-only since it is shared between multiple objects. Otherwise, it is mapped just once into virtual memory and is marked as read-write (not shared).

Both static refinements used in the custom allocator's *alloc* function (see Figure 4.7) reduce the overhead for cases where no optimization potential is expected. Compared to other OS level page sharing optimizations the optimization described in this thesis utilizes additional information about the caller of the *alloc* function and the knowledge where the R interpreter stores the frequently modified object header. For this reason, the first page of an object is not shared (see *first_page_flag* in line 16), saving the time required for the page fault handling.

After an object was allocated by the custom allocator, it may need to be copied. The duplication optimization (see Figure 4.4) augments the existing copy-on-write mechanism of the R interpreter, enabling the sharing of pages other than the global zeroed page. Lines 24 to 45 of Figure 4.7 show the pseudo-code of the optimized *duplicate* function, which changes the copy-on-write granularity to page level granularity, avoiding unnecessary copies. Similar to the allocation function, the optimized *duplicate* does not share the first page of the copy (lines 27 and 32-36) as it contains not just data but also the object's header. In line 30-44 the function iterates over the virtual pages of the original object and its copy and maps the same physical pages that are used for the original to the copy. Since these pages now need a reference count of at least two as they are mapped in both the original and the copy objects, *map_page* in line 44 automatically marks them as read-only. Additionally, in line 42 the page in the original object is also marked as read-only. Both mappings of a physical page must be marked to ensure that a write access triggers a segmentation fault that can then be used to allocate a new physical copy of the affected page. Thus, the optimized version of *duplicate* enables *lazy page allocation* for copied objects.

The dynamic refinement (as described in Section 4.3.3) that is applied to enable the deduplication of pages via a page content check is shown in the *content_check* function at the end of Figure 4.7 in lines 47 to 55. This optimization is enabled for each object that was allocated by the custom allocator and that is still alive after a garbage collection call. Each page is scanned (line 49) for zero contents that are not already shared by the global zeroed page. The global zeroed page is mapped

```
1  alloc(object_size, expect_full_write):
2    #static refinement: object size
3    if object_size < 2_pages:
4      return standard_alloc(object_size)
5
6    else:
7      #reserve a virtual address space
8      object = alloc_virtspace(object_size)
9      first_page_flag = true
10
11     for_each (virt_pg in object):
12       #the first page is always physical,
13       #static refinement: full overwrite
14       if expect_full_write or first_page_flag:
15         new_pg = get_free_physpage()
16         first_page_flag = false
17
18       else:
19         new_pg = ZEROED_PAGE
20
21       map_page(new_pg, virt_pg)
22   return object
23
24 duplicate(orig_obj):
25   #reserve a virtual address space for copy
26   copy_obj = alloc_virtspace(sizeof(orig_obj))
27   first_page_flag = true
28
29   #map virtual pages of copy to physical pages of original
30   for_each (virt_orig_pg in orig_obj,
31             virt_copy_pg in copy_obj):
32     if first_page_flag:
33       #first page is copied
34       physpg          = get_free_physpage()
35       first_page_flag = false
36       copy_content(physpg, virt_orig_pg)
37
38     else:
39       #remaining pages are just mapped to
40       #the physical pages of the original object
41           physpg = pagetable_lookup[virt_orig_pg]
42           set_readonly(virt_orig_pg)
43
44     map_page(physpg, virt_copy_pg)
45   return copy_obj
46
47 #dynamic refinement:
48 content_check():
49   for_each (pg in all_pages):
50
51     #check if the page (pg) is a copy of the global zeroed page
52     if pg != ZEROED_PAGE and page_content_is_zero(pg):
53       #map the global zeroed page instead
54       unmap_page(pg)
55       map_page(ZEROED_PAGE, pg)
```

Figure 4.7: Pseudo-code of the memory optimization core.

```
1  map_page(phys_page, virt_page):
2    pagetable_add(virt_page, phys_page)
3    remap_file_pages(virt_page, phys_page)
4
5    #update reference counter
6    refcount[phys_page] += 1
7
8    #shared pages must not be write-able
9    if refcount[phys_page] > 1:
10     set_readonly(virt_page)
11   else:
12     set_readwrite(virt_page)
13
14
15 write_fault_handler(fault_pg):
16   phys_pg = pagetable_lookup(fault_pg)
17
18   #allocate new phys. copy if page was shared
19   if refcounts[phys_pg] > 1:
20   new_pg = get_free_page()
21   copy_content(new_pg, phys_pg)
22   unmap_page(fault_pg)
23   map_page(new_pg, fault_pg)
24
25   #page is now known to be non-shared and can
26   #be used directly
27   set_readwrite(fault_pg)
```

Figure 4.8: Page mapping and page fault handler.

to the pages that have zero contents and are not already shared (line 52). The *unmap_page* function in line 54 removes the duplicated zero pages from the page table and frees the memory that was previously used by them. This function is the counterpart of the *map_page* function. Additionally, it decrements the reference counter of the previously-mapped page and marks the page as free if its reference counter is zero. The *map_page* function in line 55 then maps the global zeroed page into the virtual page occupied by the object and increases its reference counter by the number of saved pages.

The above-described functions from Figure 4.7 all use the *map_page* function. This function marks shared pages (mapped more than once into virtual memory) as read-only to trigger a function call of the *write_fault_handler* shown in Figure 4.8 (line 15–27). This fault handler allocates a new page (line 20), copies the content of the accessed page to it (line 21) and updates the virtual mapping (lines 22-23) of the page where the write access occurred (called *fault_page*). This new page is not shared yet and thus marked as read-write (line 27). Since a mapping of a virtual page is removed, the reference counter of the corresponding physical page decreases,

which is handled by the *unmap_page* function. If all but one virtual instance of a shared page were written to, the handler has created physical copies for all of these pages. The last instance, which has a reference count of one, is still marked read-only at this point because the handler only updates the pages for which a fault occurred. On a write access of this last instance the allocation and copy steps are omitted: As the reference count indicates the page is not shared anymore (line 19), it can just be marked as read-write (line 27). After the handler has mapped a new read-write page at the fault location, the interpreter can resume its execution and the write access will now succeed. Generally, such a return from a custom segmentation fault handler is not supported by the POSIX standards, but Linux and other operating systems offer extensions to allow this. Therefore, the write fault handler is implemented via the existing `libsigsegv` library [GNU18] that offers a common API for handling page faults in user mode.

In the next section, the memory savings and runtimes of the previously described memory optimization strategies are evaluated.

4.3.5 Evaluation

The results obtained by applying the proposed memory optimization strategies for R to real-world machine learning benchmarks are presented in this section. The experimental setup evaluation methodologies are described in Subsection 4.3.5. Subsection 4.3.5 discusses the results related to the memory consumption while Subsections 4.3.5 and 4.3.5 evaluate the runtime effects of the page sharing optimization strategies.

Experimental Setup

For the following experiments, a computer equipped with a 2.67 GHz Intel Core i5 M480 CPU and 6 GB of RAM, using a 64-bit version of Debian Linux 7.0 as the operating system is used. On this system, memory pages have a size of 4096 bytes. Although a larger page size than the system page size could be used for the memory optimization, the same size was chosen as it is expected to maximize the amount of memory that can be shared. (Using a smaller page size than the system size is inefficient since the optimization relies on the hardware MMU for efficient page access protection.) To evaluate the proposed memory optimization approach, the memory usage and runtime of the R interpreter including the described optimizations is compared to the standard GNU R interpreter. Both the standard as well as the optimized R interpreter are compiled using GCC version 4.7.2 with the default optimization flags (-O2) selected by the build system of R version 3.1.0.

The standard memory measurement functions for user space functions in Linux only measure the virtual memory of a process. Since the optimization approach maps the same physical page multiple times into virtual memory, these functions are not sufficient for the evaluation. They are not able to measure the amount of physical memory saved since they only count every virtual instance of a shared

physical page. Therefore a separate memory measurement function was created. To measure the amount of memory allocated by the default allocator, the standard allocation functions like `malloc` are overwritten with versions that track the current total amount of memory allocated and the original functions are called afterwards. For the optimized custom allocator, the number of physical pages that need to be reserved is directly tracked as well as the size of the memory management data structures. With these mechanisms, the amount of allocated physical memory can be measured accurately.

For the evaluation of the optimization, two different benchmark sets are applied. The first set is a shorter-running set of benchmarks, selected from the R benchmark 2.5 suite [GU08] that was originally developed to measure the performance of various configurations of the R interpreter plus one additional benchmark, listed in Table 4.1. The R benchmark 2.5 suite was also applied in other optimization approaches that focus on dynamic compilation for R [KMM+14; WWP14]. To analyze if the memory optimization is also beneficial for algorithms that already try to reduce the memory footprint by utilizing application-specific knowledge, the additional benchmark *glmnet* is included. This benchmark utilizes an existing sparse matrix optimization implemented as an R package. For accurate measurements, the iteration counts for the outer loop of each benchmark was scaled to result in a runtime of approximately 1 minute with the standard R interpreter.

Benchmark	Description
b25-1	Linear regression over a 3000x3000 matrix
b25-2	FFT of 2,400,000 random values
b25-3	Inverse of a 1600x1600 random matrix
b25-4	Grand common divisors of 400,000 pairs (recursive)
glmnet	regression using glmnet on a sparse 20000x1000 matrix

Table 4.1: Misc Benchmark Set.

The second set of benchmarks is based on a set of long-running real-world machine learning benchmarks, listed in Table 4.2. The choice of these classification algorithms is based on the method's popularity and the availability of its implementation. These algorithms were also used in Chapter 3 and are publicly available [KL18]. As in Chapter 3 the default parameters or, if available, the implementation's internal auto-tuning process was used to configure the algorithm parameters. The input data set is a 2-class classification problem and has a sufficiently large number of observations to achieve accurate results. The machine learning benchmarks were configured to use a 20-fold cross-validation. The size of the input data set (15000 samples with 200 numeric features) was chosen to ensure that the runtime of the fastest algorithms is approximately one minute on the standard interpreter. To allow for a better comparison of the memory requirements, the same data set was applied to all machine learning algorithms.

Benchmark	Description
ada	Boosting of classification trees
gbm	Gradient boosting machine
kknn	k-nearest neighbour classification
lda	Linear discriminant analysis
logreg	Logistic regression. Binary classification decision derived using a probability cutpoint of 0.5
lssvm	Least-squares support vector machine
naiveBayes	Naive Bayes classification
randomForest	Random classification forest
rda	Regularized discriminant analysis
rpart	Recursive partitioning for classification trees

Table 4.2: Machine Learning Benchmark Set.

Each benchmark was executed 10 times with the standard and the optimized version of the R interpreter. The results in the following sections are given as the arithmetic mean of these 10 executions. To reduce non-determinism, the random number seed is selected as a fixed value placed as the first statement in each of the benchmarks. Each repetition was started in a fresh interpreter process, thus initialization costs are included in the measurements (an expected overhead on the order of one second or less). The R interpreter does not use adaptive optimizations. All system services that might interfere with the measurements were disabled. Both runtime and memory usage were measured simultaneously. For these measurements, a 95% confidence interval is calculated as well as the ratio of the means using the percentile bootstrap method. Means over these ratios are calculated using the geometric mean to reduce the influence of outliers.

In the next section, the influence of the new page sharing approach on memory consumption of the selected benchmark sets is presented and discussed.

Memory Consumption

This section analyzes the benefits of the page sharing optimization techniques with regard to the memory consumption. Therefore the global peak memory usage as well as the average memory usage of each benchmarks is evaluated. The *Peak usage* represents the maximum amount of memory that was consumed during execution of a benchmark. However, the peak memory consumption does not represent information about changing memory usage over time, since the peak memory usage may occur only for an instant of time depending on the benchmark. To gain a complete view of the memory consumption the short-term peak usage is measured in intervals of 1 second, resulting in a memory-over-time profile. The *Average usage* of memory is calculated as the arithmetic mean of these 1 second measurements and used as a

second indicator besides the peak usage to allow easier comparisons of the memory behavior.

Figure 4.9 shows the peak (*Peak usage*) and average (*Average usage*) memory consumption of each benchmark running with the page sharing optimization. The 100% baseline represents the standard R interpreter without optimizations. Values below this baseline indicate relative memory savings realized by the page sharing strategies. Error bars have been omitted as the confidence intervals were smaller than 0.5% for all values.

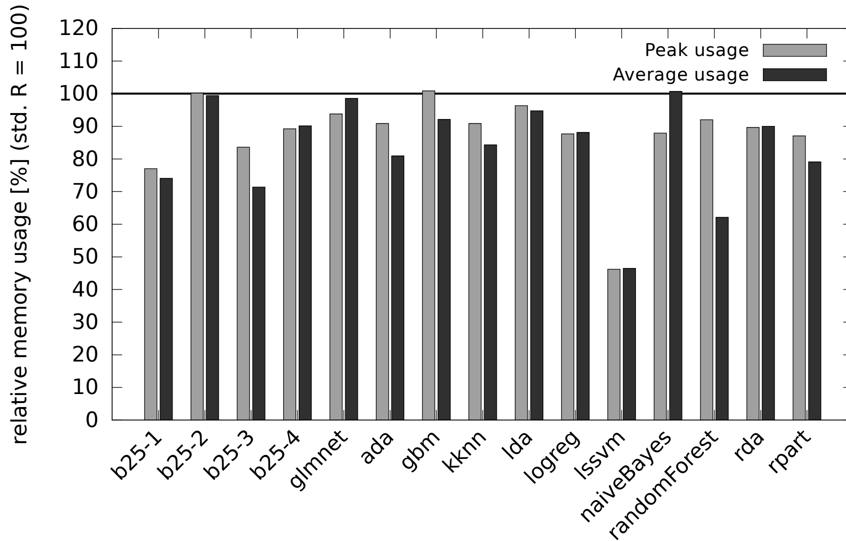

Figure 4.9: Relative memory usage with page sharing optimization compared to standard R (lower is better). The 100% baseline represents the standard R interpreter (std. R) without optimizations. Geometric means for the memory savings are 13.6% for peak and 18.0% for average memory usage.

The detailed values are presented in Table 4.3. Here, also the number of pages identified as shareable by the content check are listed since they indicate the optimization potential of the dynamic refinement (deduplication of zero pages).

The gain for reducing the peak memory usage (*GainP*) of the standard R interpreter (*Std Peak*) ranges from -0.9% for `gbm` to 53.8% for `lssvm`. However, the negative values in the columns *GainP* and *GainA* of Table 4.3 indicate that the page sharing optimizations do not gain memory savings for three of the benchmarks. Here, the peak memory consumption for two of the benchmarks (`gbm`, `b25-2`) is slightly increased as well as the average memory consumption for one benchmark (`naiveBayes`). This is caused by the additional data structures that are needed for the internal handling of memory pages.

Benchmark	Std Peak [MB]	Opt Peak [MB]	GainP [%]	Std Avg [MB]	Opt Avg [MB]	GainA [%]	ZPG [#]
b25-1	296.2	228.1	23.0	259.6	192.2	25.9	13
b25-2	131.1	131.4	-0.2	128.8	128.0	0.6	13
b25-3	197.2	164.8	16.4	157.7	112.6	28.6	37919
b25-4	134.2	119.7	10.8	127.2	114.6	9.9	194892
glmnet	354.9	332.8	6.2	249.5	246.0	1.4	46877
ada	187.2	170.1	9.1	156.0	126.2	19.1	2031992
gbm	191.5	193.2	-0.9	147.7	136.0	7.9	464
kknn	316.5	287.6	9.1	274.0	231.0	15.7	421
lda	216.2	208.2	3.7	184.8	175.1	5.3	20447
logreg	213.0	186.7	12.3	184.7	162.8	11.9	955
lssvm	1365.1	631.0	53.8	820.2	381.1	53.5	3972699
naiveBayes	143.6	126.2	12.1	80.8	81.3	-0.6	78
randomForest	565.5	520.4	8.0	390.8	242.7	37.9	1130650
rda	254.1	227.7	10.4	197.0	177.3	10.0	707
rpart	144.5	125.8	12.9	130.7	103.3	20.9	56214

Table 4.3: Memory Optimization Results: Std Peak - peak memory usage by the standard R interpreter; Opt Peak - peak memory usage by optimized interpreter; GainP - relative peak memory reduction achieved by optimization; Std Avg - average memory usage by the standard interpreter; Opt Avg - average memory usage by optimized interpreter; GainA - relative average memory reduction achieved by optimization; ZPG - number of zero pages found by the content check.

For `gbm`, still a reduction of the average memory usage by 7.9% (*GainA*) is achieved. For `naiveBayes` the situation is reversed: The optimization saves 12.1% of its peak memory usage while its average memory usage (-0.6%) is slightly increased. Since the number of pages recovered by deduplication (see column *ZPG*) is low (78), the savings of the peak memory usage are assumed to be induced by the proactive avoidance of page duplicates via the optimized allocation and duplication strategies. For `b25-2` the optimization was not able to save memory for peak memory usage and no meaningful amount for the average memory usage was saved (*GainA*). The reason why `b25-2` does not gain from the optimization is that even though it uses large vectors with 2.4 million elements, it allocates a vector which is immediately filled with random numbers similar to the pseudo-code shown in Figure 4.5. Thus, it does not profit from the optimized allocation and the content check can only recover a low number of zero pages as shown column *ZPG* (13). Furthermore, `b25-2` does not use any object duplication which is why the optimized duplication has no potential for saving memory.

Even though the page sharing optimization results in a slight increase of peak or average memory usage for the three benchmarks described above, all of the twelve other benchmarks profit from the optimization with savings in both peak and

average memory usage. The geometric mean over all fifteen benchmarks results in a reduction of peak memory usage by 13.6% and a reduction of average memory usage by 18.0%. Here, the highest amount of memory that could be saved occurs in the `lssvm` benchmark with 53.8% for peak usage and in `randomForest` with 37.9% for the average memory usage. Both of these benchmarks have a high number of zero pages recovered by the content check. Thus for those benchmarks, the reduction of the memory footprint is not just triggered by the allocation and duplication optimization but also by the dynamic refinement that deduplicates zero pages.

Table 4.3 shows summarized values for the memory consumption over the complete runtimes of all benchmarks. To gain additional insights about the memory consumption behavior, the complete profile of the memory usage over runtime will be also analyzed. Therefore, the four most interesting memory consumption profiles for the benchmarks (`glmnet`, `gbm`, `randomForest` and `naiveBayes`) are shown in Figure 4.10. For each benchmark, the run with the execution time closest to the average of its 10 executions is selected. The confidence intervals over all 10 runs of each benchmark are less than 1%, thus the figure shows only the data from a single run. The x-axis represents the runtime in seconds while the y-axis represents the corresponding memory consumption of the benchmark. Both the profile for the standard R interpreter (yellow curves) and the interpreter including the page sharing optimizations (green curves) are presented. The straight lines at the top indicate the peak memory usage, while the dotted lines mark the average memory usage.

`glmnet`: As mentioned in Section 4.3.5 the `glmnet` benchmark utilizes an already-existing memory optimization for sparse matrices. It is included in the evaluation to determine if the page sharing optimization can offer additional memory savings in the presence of an optimization that applies specialized application knowledge. In the top left of Figure 4.10 the memory-over-time behavior of this benchmark is illustrated. While there is only a small improvement for the average memory usage (see dotted green line), 6.2% of the peak memory consumption is saved (see lines on the top). The memory consumption curves show that at all local memory peaks the optimized version of the R interpreter saves a small amount of memory while the memory consumption during the remaining parts of the execution is largely unaffected. This results in only a minor reduction of the average memory consumption. Still, even in the presence of a very specific optimization for sparse matrices the page sharing optimization can offer additional memory savings. As can be seen from column *ZPG* in Table 4.3 those savings are triggered by a large number of pages recovered by deduplication (46877).

`gbm`: Not all benchmarks profit from the content checks though. For example, Table 4.3 shows that in `gbm` only 464 zero pages are recovered. This benchmark profits more from the optimizations in allocation and duplication. The corresponding memory-over-time behavior is shown in the top right of Figure 4.10. Here, the optimization does not reduce the peaks of the memory consumption, but there is a

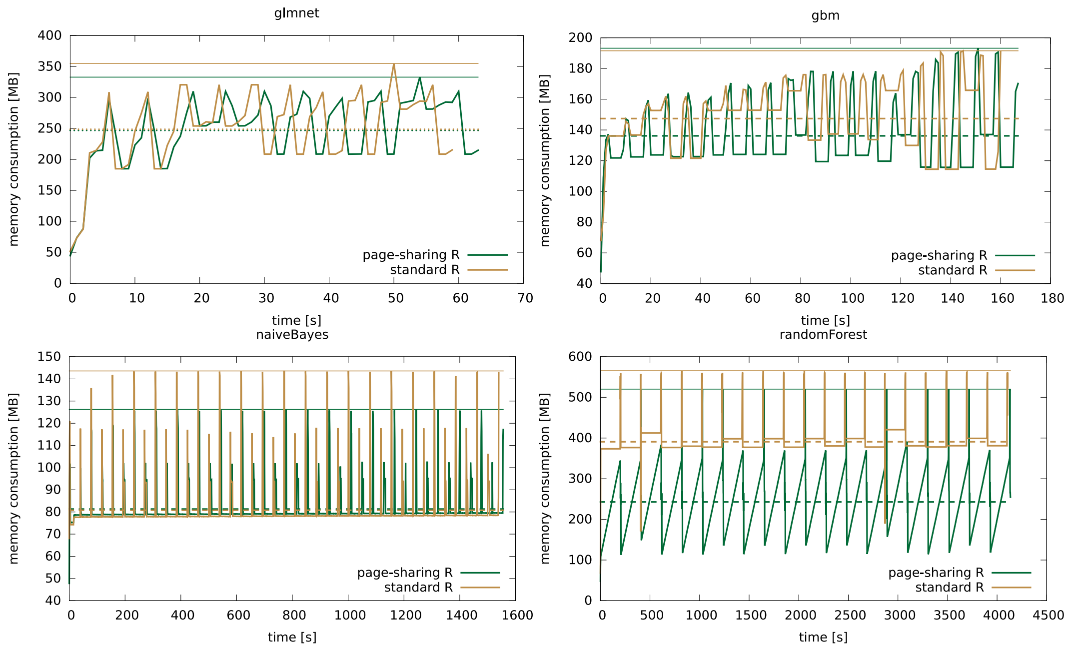

Figure 4.10: Memory consumption over time profiles for benchmarks with different memory behavior for the standard R interpreter vs. the interpreter with the page sharing optimization. Lines at the top indicate the peak memory usage, dotted lines mark the average memory usage.

marked reduction of memory usage in the valleys between the peaks, reducing the average memory consumption by 7.9%.

`naiveBayes`: Another benchmark that does not profit from the content checks is `naiveBayes` with just 78 zeroed pages recovered. Its memory-over-time profile is illustrated in the bottom left of Figure 4.10. In `naiveBayes` only the peak memory usage is reduced by the optimization (large distance between the straight lines at the top), but the average memory usage (small distance between the dotted lines) is not affected. The profile also shows that `naiveBayes` has much smaller peaks compared to `gbm`. Thus, the large reduction of memory usage at those peaks results only in a small effect on the average memory consumption.

`randomForest`: Finally, *randomForest* in the bottom right of Figure 4.10 represents one of the benchmarks where the recovery of zeroed pages triggers high memory savings. Here, the content checking reclaims 1,130,650 pages that corresponds to slightly more than 4 GB of memory. The `randomForest` profile shows a sawtooth curve for the optimized interpreter (see green curve). This indicates that the benchmark uses large blocks of memory that are slowly written to. For the page sharing optimizations, this represents an ideal memory usage pattern as the allocation of memory is delayed until the benchmark writes data to it. This results in a 37.9% improvement of the average memory consumption (large distance

between dotted lines) – the average time during which the benchmark has a high memory consumption is reduced.

Looking back at the profile of `glmnet` (top left), the green curve which shows the profile for the optimized interpreter is longer than the yellow curve for the standard interpreter and there is an increasing shift between the peaks of both curves over time. The reason for this lies in the additional CPU time needed to provide the page sharing optimizations. The runtime overhead induced by the memory optimization will be analyzed in more detail in the next section.

Runtime Overhead

There are multiple reasons for the runtime overhead caused by the optimization: First, in the optimized allocation both virtual and physical pages need to be managed. The static refinement (as described in Section 4.3.2) that disables the custom allocator for objects above a size limit of two pages avoids this overhead for small objects where the optimizations potential is low.

Second, the first write access to a shared page requires intervention of the fault handler, which also induces runtime overhead. To reduce this overhead another static refinement that excludes functions which are known to immediately overwrite their memory allocation is applied. However, this refinement only covers the case where both the allocation and write access happen in the same function. If the immediate write after allocation is caused by calculations from different functions the benchmark, the static refinement is not able to detect this.

Third, the dynamic refinement that checks the content of pages for deduplication also adds runtime overhead. Its overhead is expected to be low since it only checks for pages which contain zero bytes.

Table 4.4 shows the execution times of the benchmarks for both the unoptimized standard (*Std*) R interpreter and the interpreter including the optimization (*Opt*). The relative overhead of the optimization is given in the column *Loss* and the number of times the content check optimization was triggered is given in column *CC*. Here, the confidence intervals have been omitted as they were smaller than 1.0% .

The highest amount of runtime overhead is caused in the `b25-4` benchmark (16.9%). Here, the immediate write access after allocation does not happen in the same function and thus can not be detected by the static refinement. `b25-4` recursively calculates the greatest common divisors for two vectors. The R interpreter duplicates those vectors passed as function parameters in each recursion and the benchmark updates these vectors. The overhead of the content check is expected to be low for benchmarks where it does not contribute significantly to the memory savings. An example of a low overhead for content checks is represented by the `ada` benchmark with 1.8%. Here, the content check was triggered 12113 times (see column *CC*) resulting in over two million recovered zero pages according to Table 4.3. On the other hand, in the `lssvm` benchmark, with an overhead of 13.3%, just 1002 content checks were performed that recovered almost four million zero pages.

Benchmark	Std [s]	Opt [s]	Loss [%]	CC [#]
b25-1	69.3	69.8	0.8	10
b25-2	61.6	62.0	0.7	58
b25-3	57.2	58.4	1.9	23
b25-4	60.2	70.3	16.9	1045
glmnet	59.3	64.0	7.9	156
ada	9874.3	10055.7	1.8	12113
gbm	160.4	168.0	4.7	287
kknn	2030.8	2063.6	1.6	486
lda	86.5	98.1	13.3	334
logreg	82.8	89.6	8.2	280
lssvm	530.5	601.2	13.3	1002
naiveBayes	1539.7	1555.6	1.0	31466
randomForest	4107.1	4135.5	0.7	224
rda	7617.8	7871.8	3.3	11849
rpart	61.5	64.2	4.4	304

Table 4.4: Runtime Results: Std - runtime with standard R interpreter; Opt - runtime with optimized R interpreter; Loss - relative runtime overhead incurred by optimized R interpreter; CC - number of executed content checks; Geometric mean of runtime loss is 5.3%; Confidence intervals have been omitted as they were smaller than 1.0% for all values.

This shows that the overhead of the content check depends not just on the number of times it is triggered but also on the number of pages it has to search through.

Overall, the runtime overhead for all benchmarks is just 5.3%.

The static refinement that checks the object size to determine if the custom allocator should be enabled provides the opportunity to modify the size limit in order to change the runtime overhead. Here, an increase of the object size limit is expected to result in a decrease of the number of objects that are considered for optimization which results in a trade-off between memory savings and runtime overhead.

This trade-off is illustrated in Figure 4.11 for the `logreg` benchmark. The x-axis shows five different object size limits for the custom allocator. The y-axis represents the runtime (grey) and peak memory consumption (black) of the optimization relative to the same values for the standard R interpreter. Error bars have been omitted as the confidence intervals were smaller than 0.3% for all values. For both memory and runtime, a lower percentage is better, since the 100% baseline represents the values of the standard R interpreter. The 8 KB limit is the limit that was chosen in the previously presented measurements.

The results show that for `logreg` increasing the size limit for custom-allocated objects reduces the gain of the optimizations as fewer objects can be optimized. With a size limit of 1024 KB, the gain is even slightly negative as the optimization

induces some memory overhead due to additional data structures needed for the page sharing. This shows that for the `logreg` benchmark memory savings are only triggered by R object allocations with a size smaller than 1024 KB. Furthermore, the increase of the size limit reduces the number of objects handled by the optimization thus the overhead contributes less to the overall runtime of the benchmark.

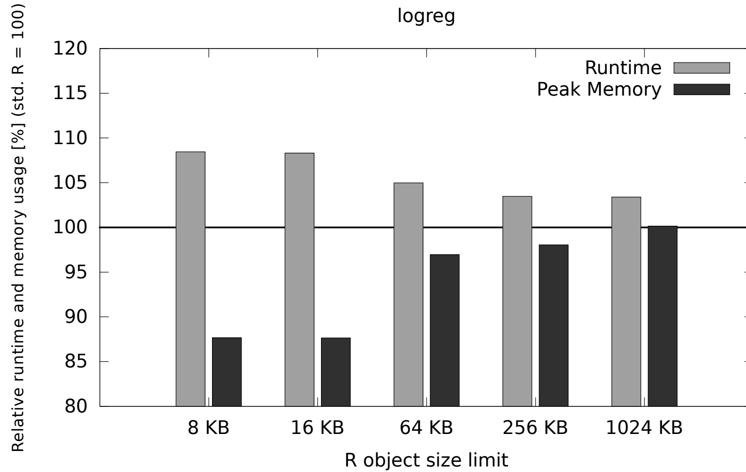

Figure 4.11: Static refinement using different object size limits; Error bars have been omitted as the confidence intervals were smaller than 0.3% for all values (lower is better).

In all previous measurements, the RAM available in the system was sufficient to hold all data used by the benchmark. If this is not the case, the above-discussed runtime overhead can become insignificant which will be illustrated in the next section.

Runtime Reduction

When the amount of RAM in the system is too small to hold all data required by the benchmark, there are situations where the proposed memory optimization is also able to reduce the runtime of the benchmark instead of adding overhead. This is due to frequent page swaps that require disk I/O when the total capacity of RAM is exceeded, a phenomenon also known as "thrashing". To analyze this situation, two benchmarks are considered. The first one is the `lssvm` benchmark where the optimization provides a large reduction in memory consumption. The second benchmark is an instance of `logreg` where the optimization provides only smaller memory gains.

For the analysis the memory requirements of the benchmarks need to be increased beyond the capacity of the RAM in the system. Instead of increasing the data set size of both benchmarks, the system is limited to just 1 GB of RAM instead,

Benchmark	Std Peak [MB]	Opt Peak [MB]	Gain Peak [%]	Std Avg [MB]	Opt Avg [MB]	Gain Avg [%]
logreg-2 1 GB	1228.2	1094.8	10.9	965.7	789.6	18.2
logreg-2 6 GB	1228.2	1094.8	10.9	967.8	823.2	14.9
lssvm 1 GB	1365.1	631.1	53.8	970.0	381.3	60.7
lssvm 6 GB	1365.1	631.0	53.8	820.2	381.1	53.5

Benchmark	Std [s]	Opt [s]	Speedup (CI)
logreg-2 1 GB	6395.5	5785.6	$1.105_{1.071}^{1.144}$
logreg-2 6 GB	579.8	598.5	$0.969_{0.967}^{0.971}$
lssvm 1 GB	3080.3	593.8	$5.188_{5.029}^{5.350}$
lssvm 6 GB	530.5	601.2	$0.882_{0.880}^{0.885}$

Table 4.5: Evaluation results with two configurations of RAM; see Table 4.3 and 4.4 for column descriptions, except Speedup: Runtime speedup factor (Std / Opt). Confidence intervals for runtime are shown (CI), others have been omitted as they are smaller than 0.8%.

since the runtimes of the benchmarks do not scale linearly with the data set size, leading to excessively high execution times. However, since the `logreg` benchmark has a much smaller memory consumption than 1 GB, the data set size for `logreg` is additionally increased to 70000 samples with 300 numeric features, which increases the memory requirements of this benchmark to approximately the same level as `lssvm`. This still results in acceptable execution times for `logreg`.

Table 4.5 shows the results for the previous 6 GB system configuration and the limited 1 GB RAM configuration for both benchmarks. The `logreg` benchmark is now shown as `logreg-2` because it was executed with the previously described larger data set. In the 1 GB configuration, the system had to swap for both the standard and optimized interpreters, resulting in a large increase in runtime over to the 6 GB configuration. The peak memory usage for the interpreters is identical in both configurations while the average memory usage differs as this value is time-dependent and thus influenced by swapping. This swapping also increases the variability in the runtime measurements, thus the confidence intervals for the speedup factors are also included (see lower part of Table 4.5).

Reducing the available memory from 6 GB to 1 GB has drastically increased the runtime for both versions the standard R interpreter (*Std*) as well as the interpreter including the memory optimization (*Opt*). Still, the reduction in memory consumption for `logreg-2` has turned the slowdown (factor 0.969) in its 6 GB configuration into a small speedup (factor 1.105) when the RAM is limited to 1 GB. Depending on the benchmark and its memory usage pattern, a different situation could also happen: In the worst case, the content check of the optimized interpreter touches a large number of pages, forcing them to be swapped in. This additional

swap activity can increase the runtime so that the gains from a reduced memory footprint may become irrelevant.

The second benchmark `lssvm` shows something closer to the best case for the optimization: Here, the page sharing optimization manages to save enough memory to avoid swapping. In this case, significant speedups are gained as shown in the lower part of Table 4.5 for the 1 GB configuration of `lssvm`.

Similar to `logreg-2`, the memory usages do not vary much between both configurations (see upper part of Table 4.5). Considering the runtime results, the optimized interpreter *Opt* only needs 593.8 seconds to run the `lssvm` benchmark which is almost unchanged from the 6 GB configuration (601.2 seconds). On the other hand, the standard interpreter *Std* has now increased its runtime to 3080.3 seconds (51.3 min.) when limited to 1 GB of RAM. This makes the overhead of the memory optimization irrelevant as the time gained by avoiding page I/Os is much larger. The page sharing optimization enables a speed up by a factor of 5.2 for `llsvm` by reducing the peak memory consumption by 53.8%. This speed up is also illustrated in Figure 4.12 that shows the memory consumption profile for one exemplary execution of the `lssvm` benchmark.

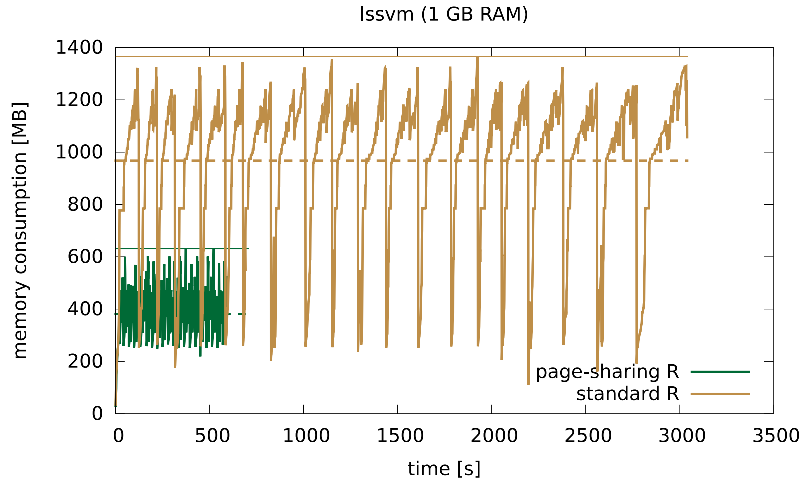

Figure 4.12: Exemplary memory consumption over time profile for the lssvm benchmark. Reaching a speed up by a factor of 5.2 on a system with 1GB of RAM. Lines indicate the peak memory usage, dotted lines mark the average memory usage.

This shows that reducing the memory consumption with the page sharing optimization can significantly improve the runtime for memory–hungry benchmarks if the available RAM is constrained. In turn, this can enable the processing of larger data sets.

4.4 Summary

In the domain of machine learning algorithms, the R interpreter induces a large memory overhead due to wasteful memory allocation policies [KKL+14]. The goal of the presented memory optimization approach was to enable efficient memory utilization, especially for memory-hungry R applications like machine learning algorithms. To accomplish this goal, this chapter presented an application-transparent memory optimization approach that employs page sharing at a memory management layer between the R interpreter and the operating system's memory management. The optimization benefits a large number of applications since it preserves compatibility with the available software libraries that most R programs are based on, and covers one of the most important resource bottlenecks of machine learning algorithms. By concentrating on the most rewarding optimizations – the sharing of zero-filled pages and deduplicating at the page level instead of the object level –, the overhead of more general OS level memory optimization approaches such as deduplication and compression is avoided. With the proposed optimization, considerable reductions of the memory consumption for a large number of typical real-world benchmarks have been achieved, which is an important step towards processing larger input sizes. It also significantly speeds up the computation in cases where previously pages had to be swapped out due to insufficient main memory.

Currently, the optimization approach has to replicate a subset of the virtual memory information that is already available in the OS kernel. Korb et al. [KKM16] developed a hybrid user level/kernel mode implementation of the presented page sharing approach, which results in a small reduction of the overhead of the current user mode-only implementation. The optimizations are based on a new system call that uses the existing copy-on-write functionality of the Linux kernel to avoid duplicating memory when data is copied. For a further reduction of the overhead, machine-learning techniques could be applied to optimize the trade-off between runtime and memory, since the parameters of the presented approach allow for dynamic tuning. However, this is outside the scope of this thesis and left as a direction for future work.

In addition to the proposed memory optimization, the second major avenue for speeding up statistical machine learning algorithms that we will explore in this thesis is parallelization. Parallelizing the execution poses new resource utilization challenges. In order to fully benefit from parallel execution, it is essential to analyze the resource utilization of the parallelized version of these learning algorithms. The next chapter therefore presents a performance analysis of parallel machine learning algorithms, with the goal to develop new resource-aware mapping and scheduling methods.

Chapter 5

Profiling of Parallel Machine Learning Algorithms

This chapter presents an analysis of parallel machine learning algorithms implemented in the R programming language and is based on the papers by Kotthaus et al. [KKM15a; KKM15b]. As outlined in Chapter 2, the GNU R language is the most widely used programming language for statistical data analysis. While apparently not affecting the popularity of the R language, its lavish use of resources makes it unsuitable in an environment where high performance is required, or where resources are scarce. Since R is increasingly used to process large data sets, parallelization is used to speed up computation, which poses new resource utilization challenges. Building upon the analysis of bottlenecks arising during parallel execution, the optimization potential for resource utilization can be estimated. To fully benefit from multi-core architectures, it is therefore essential that the analysis is not restricted to single-threaded algorithms, but also includes parallel learning algorithms and their mapping and scheduling on underlying architectures.

This chapter presents the parallel profiling mechanisms of the traceR profiling framework [tra18] and the analysis of bottlenecks arising in embarrassingly parallel machine learning applications. As presented in Chapter 3, traceR allows for profiling the resource usage of an R application to locate bottlenecks and develop new optimizations [KKL+14; KKK+14]. Its profiling functionality was previously limited to sequential R applications. Profiling parallel R applications therefore required improving the tool with new profiling mechanisms.

The analysis with traceR focuses on algorithms that consist of embarrassingly parallel independent tasks. While there are many different parallel machine learning algorithms, the analysis in this chapter will mainly focus on hyperparameter optimization for machine learning algorithms, which is an algorithm with huge resource demands. The goal of *hyperparameter tuning* is to find the best parameter configuration for building a model for the given data in a limited time budget. Evaluating these configurations usually requires a huge amount of resources. The goal of the analysis is to support the development of new optimization methods for parallel machine learning applications, including mapping and scheduling methods with respect to the resource utilization.

This chapter is structured as follows: The fundamentals of parallel execution mechanisms of the R language are introduced in Section 5.1. Section 5.2 describes the parallel profiling mechanisms of the traceR profiling framework that serve as

a basis for the analysis. Section 5.3 presents the runtime analysis and examines the memory consumption and CPU utilization of parallel executed hyperparameter tuning. Finally, Section 5.4 concludes with a summary of the results.

5.1 Parallel Execution Mechanisms in R - Fundamentals

The R language has a *single-threaded design* and is not thread-safe. Since 2012 it however contains the ability of parallelizing full processes, e.g., via the *fork* mechanism. Furthermore, R can be extended with external *multi-threaded libraries* like the OpenBLAS library [ZWW18] used for basic vector or matrix operations, or with packages that use threading (e.g., pthreads [IE94], OpenMP [DM98]) in their external C code parts [R C17a].

The CRAN task view for high-performance and parallel computing with R[1] gives an overview of all packages that support parallel computing within R. Figure 5.1 visualizes this broad range of parallel execution mechanisms supported via R packages.

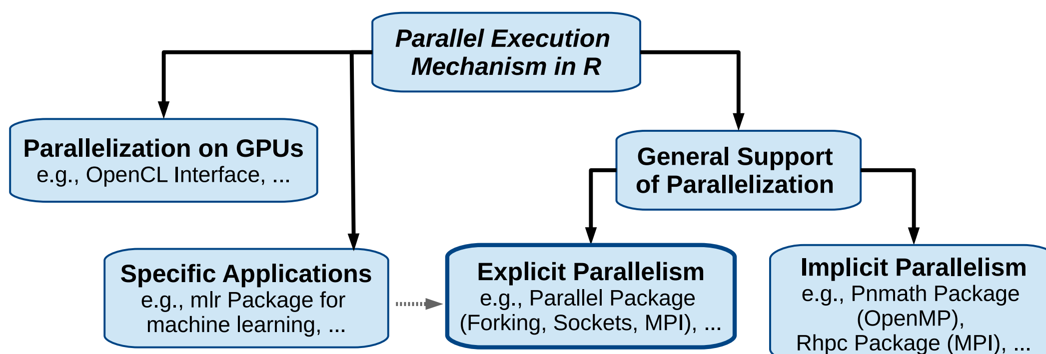

Figure 5.1: Parallel execution mechanisms in R supported via packages.

Besides packages for parallelization on *GPUs*, there are several libraries that implement parallelism for *specific applications*. Some of those specific applications exploit the general parallel execution mechanism that R supports by default. From the user perspective, there are two ways to incorporate parallelism in an application: *implicit* and *explicit*. Implicit means that the parallelization is transparent to the user, while explicit requires the user to manage the parallelization, including the task and data distribution and partitioning.

This thesis solely focuses on explicit parallelization strategies. Therefore, the following parts of this chapter will focus on parallel applications consisting of computation intensive independent tasks that do not need to communicate and that

[1] CRAN Task View: High-Performance and Parallel Computing: https://cran.r-project.org/web/views/HighPerformanceComputing.html, 2018

utilize the basic parallel mechanisms of R via the `parallel` package [R C17a]. The parallel package was included in the R language in 2012 for the direct support of parallel computation and is thus the most common kind of explicit parallelism used. A common parallel application is to execute the same function on different input data sets. Therefore, two different basic computation models are offered:

Prescheduling: The first computation model is called *prescheduling*. It is recommended when the parallel tasks (jobs) have similar execution times. In this model, so-called *worker* processes are started (e.g., one worker per available CPU). The parallel application that runs the same computation on different data sets in parallel is split into equal-sized sets of jobs, where the number of sets should equal the number of worker processes. Those sets of jobs are then sent to the worker processes. A master process waits for all jobs to finish and then terminates the workers.

Load balancing: The second parallel computation model is called *load balancing* and is recommended when jobs have widely varying execution times. Instead of pre-splitting the jobs into equal-sized sets, the master process consecutively sends a single job to each worker process. It then waits for any worker to finish its job, and sends the next job to the next idle worker, until all jobs are completed.

In the next section, the parallel profiling mechanisms of the traceR framework are presented that will later serve as a basis for the analysis. Both of the above-described parallel computation models will be analyzed.

5.2 Parallel Profiling Mechanisms - traceR

The traceR profiling tool [tra18] that was presented in Section 3.2 can also be utilized for examining parallel R applications. It supports common cases like profiling of parallelization on multiple cores (Subsection 5.2.1) or multiple machines (Subsection 5.2.2). The parallel profiling mechanisms enable the analysis of the CPU utilization of parallel tasks and the memory usage of parallel programs during execution.

Since the gain from parallel execution can turn into a heavy penalty if the memory requirements of all parallel processes exceed the capacity of the system, causing thrashing, the profiling data can help determine the maximum degree of parallelism. The data can furthermore be utilized to detect inefficient resource utilization and guide scheduling decisions with respect to the resource utilization and thus supports the development of new optimizations for parallel applications.

This section illustrates the mentioned profiling mechanisms on an example program for calculating a Mandelbrot fractal in parallel. Those mechanisms will later serve as a basis for the profiling of parallel machine learning algorithms in Sections 5.3.

5.2.1 Single Machine Profile

The traceR framework produces profiles that visualize the relative CPU utilization and memory consumption for applications executed in parallel. Additionally, traceR also generates more detailed runtime and memory profiles for each parallel process separately [tra18]. This section demonstrates the profiling mechanisms of the traceR framework on a single machine with multiple cores. For this purpose, a parallelized R program for calculating a Mandelbrot fractal image is used as an example application, profiled on a Lenovo L512 notebook with an Intel Core i5 (dual-core, 2nd Gen) processor with 2.5 GHz and hyper-threading, running R version 3 on Linux.

To execute the Mandelbrot fractal calculation in parallel, the computation is partitioned into jobs, each calculating a different block of lines of the final image. The so-called `mclappy` function of the parallel package [R C17a] is used, which by default uses the prescheduling mechanism as described in Section 5.1. The mclapply function first splits the jobs into as many sets as the number of cores specified, and then sends them to the worker processes, each covering more than one job.

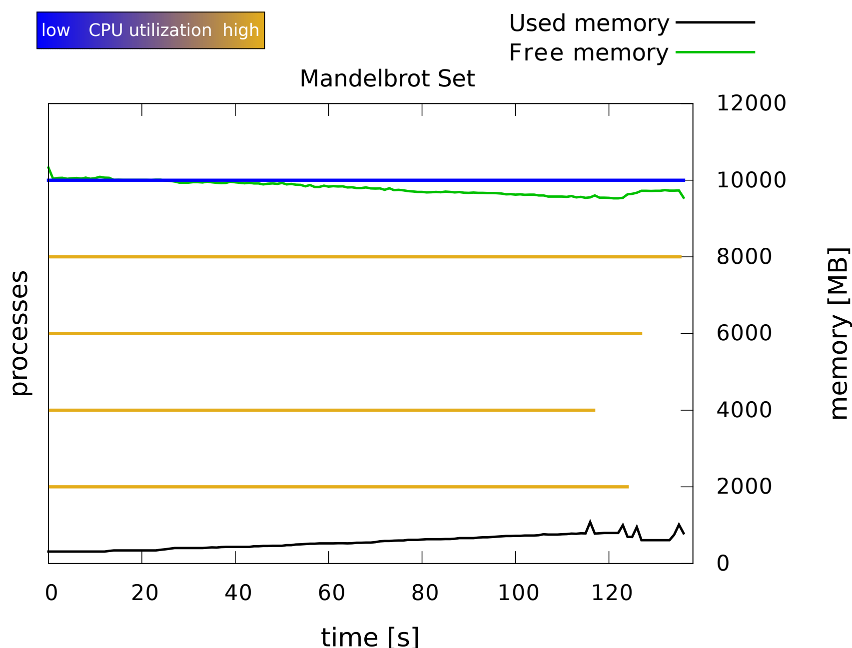

Figure 5.2: Relative CPU utilization and memory consumption of a parallel R program for calculating a Mandelbrot fractal on 4 cores on a single machine with prescheduling.

Figure 5.2 shows the traceR profile for the described Mandelbrot fractal calculation conducted on four cores. This kind of profile will also be later used in the analysis of parallel machine learning applications. The y-axis on the right side shows the memory utilization of the Mandelbrot fractal computation. The memory

behavior is described by two curves. The green curve shows the amount of free main memory during program execution, as reported by the Linux kernel. The black curve shows the amount of memory allocated by all parallel processes over time. The values for the free and the allocated memory are sampled once per second. The x-axis represents the runtime in seconds. The runtime and the CPU utilization of the parallel processes are illustrated by horizontal lines. The length of a line represents the runtime. Here, the color of the line can vary from blue via purple to orange and indicates the relative CPU utilization of a process, averaged over its entire runtime.

The measurements for the master process shown in the first horizontal line indicate a low CPU utilization (blue) since the master only waits for the workers to finish and gathers the results at the end without running any calculations.

Since `mclapply` uses one worker process per core, Figure 5.2 shows four orange lines for the worker processes. The worker processes are orange since their CPU utilization is high. This indicates that they did not have to wait for CPU resources instead, they received the maximum amount of CPU time. If their color would shift to purple or blue this would indicate that they had consumed less CPU time than their wall-clock runtime. This could be caused by other concurrent processes running on the same machine, by running more processes than CPU cores available or just by processing delays that are for example caused by I/O operations.

In the above-described example, the total runtime of the Mandelbrot fractal calculation was reduced by executing it on four cores in parallel. To further reduce the runtime, a common way is to distribute the execution across multiple machines that together may comprise even more cores. This scenario will be described in the next section.

5.2.2 Multiple Machine Profile

Besides analyzing the memory consumption and CPU utilization on a single machine, traceR is also able to profile programs running on multiple machines in a distributed fashion. To demonstrate this profiling mechanism, the Mandelbrot fractal calculation of the previous section is used. For simplicity, we assume that two machines with four cores each are used. For distributed execution, R's parallel package offers various functionalities like, e.g., the `makePSOCKcluster` function. This function launches copies of the R interpreter on specified machines (hosts) to set up worker processes which listen on a socket for evaluation jobs via the *Rscript* front end, an alternative front end for using R in scripts. These machines together form a "cluster".

When the calculation of a program is distributed across multiple hosts, traceR creates one sub-profile for each host. In this case, the master process is only shown in the sub-profile of the machine it was executed on. Figure 5.3 presents the traceR profile for calculating the Mandelbrot image on two hosts and eight cores. Compared to the runtime (approximately 140 seconds) shown in the profile of Figure 5.2 where

the program was executed on four cores, only minor runtime savings were gained when calculating the Mandelbrot image on eight cores (approximately 120 seconds, i.e., 20 seconds less).

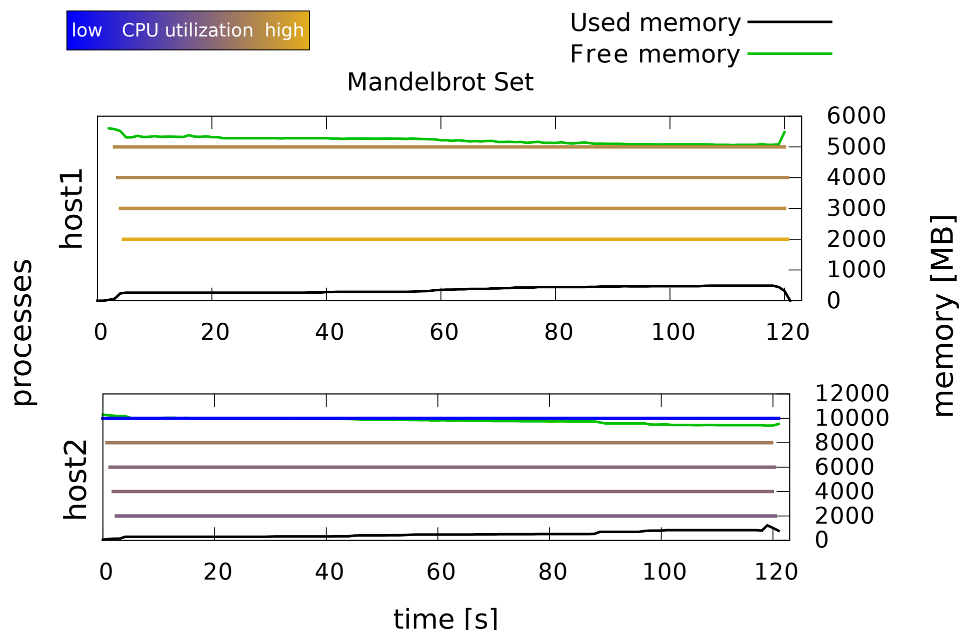

Figure 5.3: Relative CPU utilization and memory consumption of an R program for calculating a Mandelbrot fractal using 8 cores on 2 machines.

This indicates an inefficient resource utilization. One reason for this is the overhead of using Rscript to launch copies of the processes. The CPU utilization of the worker processes on *host2* where the master process is executed (see blue line in lower part of Figure 5.3) suggests another reason. In the previous profile where the program was executed on a single machine, all worker processes had a high CPU utilization and also a variance in completion times. Now the individual runtimes of the worker processes are very similar to each other and their CPU utilization is lower (purple colored lines). The reason for this is that the worker processes are not able to stop execution when their computation is finished, they have to wait for the master process and the master first waits for all computations to be finished before it shuts down the worker processes.

In summary, using two machines for the Mandelbrot example produces excessive overhead and thus does not lead to an efficient runtime reduction. A single machine with more cores might have yielded a higher speed-up due to reduced overhead.

The next section analyzes the parallel runtime performance of machine learning algorithms based on the presented profiling mechanisms.

5.3 Profiling of Parallel Machine Learning Algorithms

The goal of the analysis in this section is to support the development of new optimization methods for parallel machine learning applications, including mapping and scheduling methods with respect to the resource utilization. While there are many different parallel machine learning algorithms, the analysis in this section will focus on *hyperparameter tuning of machine learning algorithms*, which is an algorithm with huge resource demands. The goal of hyperparameter tuning is to find the best parameter configuration for building a model for the given data in a limited time budget. To speed up the tuning process, the configurations are evaluated in parallel, which typically requires a large amount of resources.

In the following, the runtime behavior (Subsection 5.3.2) and the memory consumption together with the CPU utilization (Subsection 5.3.3) of a real-world application for tuning the hyperparameters of a *Support Vector Machine* (SVM) classification are analyzed.

5.3.1 Experimental Setup

The classification performance of a SVM highly depends on its parameter configurations, therefore it is important to tune those parameters by evaluating the prediction performance of different configurations. Here, different parameter configurations are evaluated, based on the `mlr` R library [BLK+16]. The configurations are selected via a random search and evaluated in parallel. The parameters γ and cost C are both box-constrained to the interval $[-15, 15]$ on a $\log_2$-scale. The SVM implementation is based on `libsvm`[2] and implemented in the R library `e1071` [ADK06]. The data set `w8a`[3] is used as the input for the SVM optimization. The prediction performance of automated parameter tuning algorithms will be analyzed in Chapter 6. Therefore, the analysis in this chapter solely focuses on the resource utilization performance of parallel execution.

The mlr library utilizes the R function `mclapply` of the parallel R package [R C17a], which is configurable with two different parallel computation models: prescheduling and load balancing (as described in Section 5.1). Both configurations will be examined. The profiling of the SVM classification is conducted on a Lenovo L512 notebook with an Intel Core i5 (dual-core, 2nd Gen) processor with 2.5 GHz and hyper-threading, running R version 3 on Linux.

5.3.2 Runtime Behavior Analysis

In the following, the runtime behavior of the parallelized hyperparameter tuning algorithm described above is analyzed with traceR. The SVM classification configurations are evaluated on a single machine using four cores and 4 GB of main memory.

[2] Chang, C. et. al: `https://www.csie.ntu.edu.tw/~cjlin/libsvm`
[3] Platt, J.C.: `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/w8a`

First, the runtime behavior of the *prescheduling* parallel computation mechanism is examined, and then the *load balancing* model is analyzed.

Prescheduling Mechanism

When the default prescheduling mechanism of the parallel package is used, a pool of worker processes is created. The master process is replicated via fork and the overall set of jobs is split into smaller sets, each of which is assigned to a single worker Each job represents the evaluation of the performance of the SVM with a specific parameter configuration. Figure 5.4 presents the traceR profile of the parallel SVM parameter tuning benchmark with prescheduling (as described in Section 5.2.1).

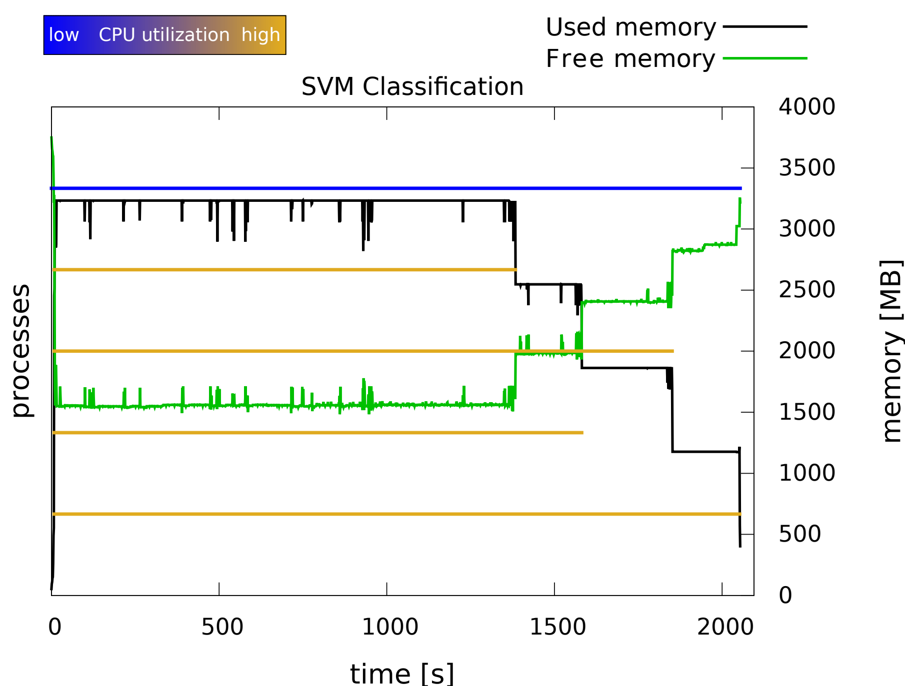

Figure 5.4: CPU utilization and memory consumption of an R program evaluating different parameter configurations of a SVM classification using 4 cores on a single machine with the default prescheduling mechanism.

The measurements for the master process shown as the first horizontal line indicate a low CPU utilization (blue) since the master only waits for the workers to finish and gathers the results at the end without running any calculations. The four orange lines represent the runtime and the CPU utilization of the worker processes. The worker processes are orange and thus have a high CPU utilization, which indicates that they did not have to wait for CPU resources, instead they received close to an entire CPU core each.

However, the completion times of the worker processes have a high variance, which is indicated by the differing lengths of the horizontal lines at the end of

the execution. The default computation model of prescheduling hence produces an inefficient resource utilization for the SVM application. The parallel package supports a mechanism for balancing jobs with high runtime variances more evenly, which will be analyzed in the following.

Load Balancing Mechanism

Load balancing is recommended if the jobs of a parallel program have varying completion times or if the underlying architecture is heterogeneous, e.g., different CPU cores having different frequencies. Figure 5.5 presents the traceR profile for CPU utilization and memory consumption of the SVM parameter tuning program, now executed with the load balancing mechanism. To visualize the load balancing option, the profile shows one horizontal line for the CPU utilization of each job instead of each worker process. All jobs are still processed on four cores.

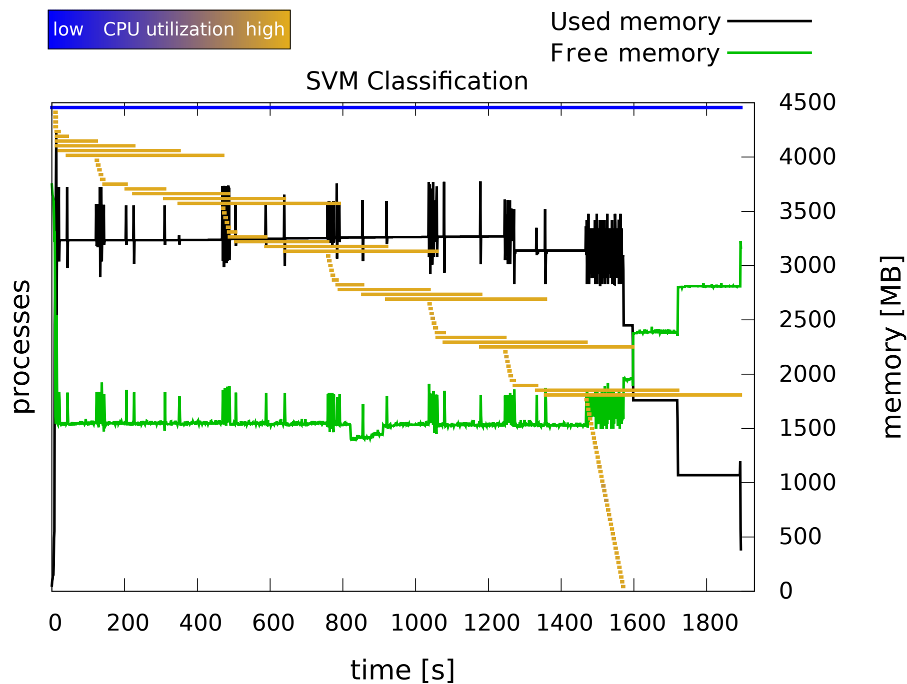

Figure 5.5: CPU utilization and memory consumption of an R program evaluating different parameter configurations of a SVM classification using 4 cores on a single machine with the load balancing mechanism.

Compared to the profile from Figure 5.4, where prescheduling was activated instead of load balancing, the dynamic allocation of jobs to workers did not translate into major runtime savings (approximately 160 seconds). This is due to the fact that there still is a high variance of completion times at the end of the program, leading to an inefficient resource utilization. This shows that neither the load balancing

nor the prescheduling mechanism is sufficient for achieving an efficient utilization of resources for the SVM parameter tuning algorithm. However, a high variance of execution times is not specific to this SVM example, but is a common case for parameter tuning of machine learning algorithms. This calls for the development of new, more resource-efficient parallelization strategies.

Inefficient resource utilization can not only be caused by insufficient load balancing. In the next section, an example case where it is caused by creating too many parallel processes is discussed.

5.3.3 Memory Consumption and CPU Utilization Analysis

The gain from parallel execution can be negated if the memory requirements of all processes running in parallel exceed the capacity of the main memory. In this situation, the operating system starts to swap out memory which dramatically slows down the execution.

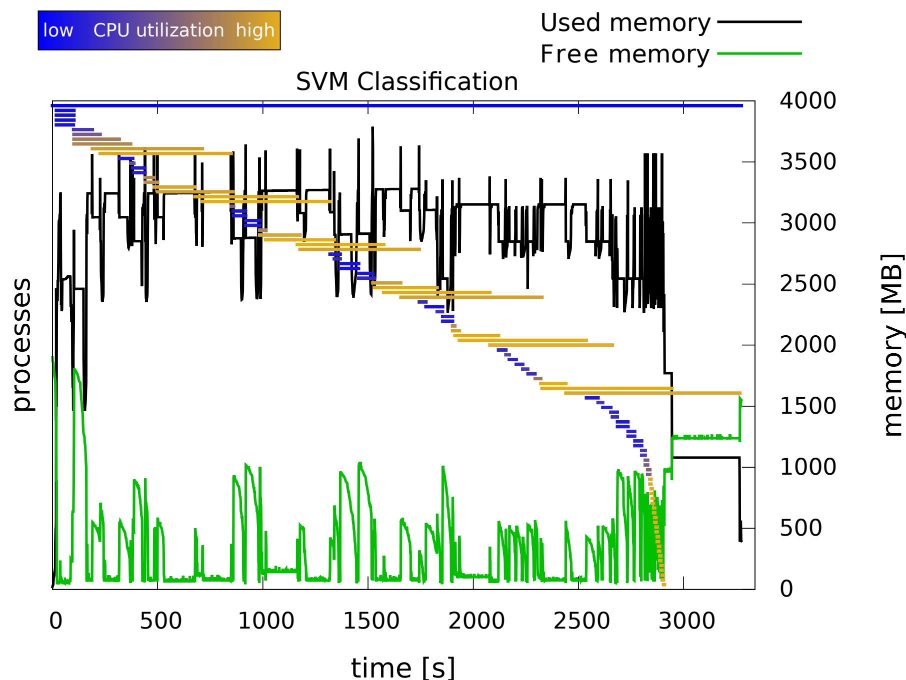

Figure 5.6: CPU utilization and memory consumption of an R program evaluating different parameter configurations of a SVM classification using load balancing on 4 cores on a single machine with 2 GB of main memory.

The traceR profile in Figure 5.6 illustrates this scenario. Here, the SVM parameter tuning program was profiled running with lower memory resources of 2 GB of main memory instead of 4 GB.

In comparison to Figure 5.5, the system runs out of free main memory, indicated by the green curve that in some phases of the execution is close to zero. This also

leads to a high delay in execution time as indicated by the total runtime on the x-axis. Furthermore, this profile includes blue lines indicating a low CPU utilization.

Since the parallelly executed evaluations of the SVM parameter configurations are all independent, they are not aware of each others memory usage, thus one job cannot trigger garbage collection in another and free memory for it. This example demonstrates that the execution of too many parallel processes leads to wasted resources due to inefficient allocation of resources. The profiling data produced by the traceR framework thus can serve as an indicator to determine the maximally feasible degree of parallelism.

5.4 Summary

This chapter summarized the parallel execution mechanisms included in the R programming language and presented the parallel profiling mechanisms of the traceR profiling framework that were developed in this thesis. The profiling mechanisms were used for analyzing the bottlenecks of a parallelized machine learning application.

The problem of high runtime variance of parallel processes, even when using the load balancing option of the R runtime environment, was examined and the issue of executing too many parallel processes leading to inefficient resource utilization was discussed. The results of the analysis show that the parallel computation models of the R language are not sufficient for machine learning applications like hyperparameter tuning, as they do not enable an efficient utilization of resources.

The results of the analyses gathered by traceR can guide the development of new optimization methods for parallel machine learning applications, including new scheduling strategies with respect to resource utilization. Such strategies are especially important if the system architecture is heterogeneous or if the parallelly executed jobs of an application have varying resource requirements depending on the input data. Overall, the results of the analysis call for the development of new resource-efficient parallelization strategies. Chapter 6 of this thesis therefore concentrates on the development of optimized scheduling strategies for parallel machine learning applications.

Chapter 6

Resource-Aware Scheduling Strategies for Parallel Machine Learning Algorithms

This chapter presents resource-aware scheduling strategies for parallel machine learning algorithms based on the papers by Kotthaus et al. [RKB+16; KRL+16; KRL+17; KLN+17]. While there are many different parallel machine learning algorithms, we will focus solely on *Model-Based Optimization (MBO)*, which is a learning algorithm with huge resource demands. As described in Chapter 2, MBO is a state-of-the-art global optimization method for black-box functions that are expensive to evaluate. A black-box function is an abstraction of a system for which the internal working is unknown, hence the only way to establish a relationship between input parameters and output values is to evaluate it at each respective point in the input space. One of its uses is hyperparameter tuning of machine learning algorithms where the black-box is a machine learning algorithm [THH+13; LKM+15]. The goal is to find the parameter configuration of the algorithm with the highest quality of the output measured by a given performance criterion within a limited time budget. Due to huge model spaces (many different parameter configurations), a large amount of resources is needed to evaluate these configurations [CVB+02]. Here, resource requirements like CPU utilization or memory usage vary heavily depending on the type and configuration of the applied machine learning algorithm.

Figure 6.1 visualizes the *heterogeneity of the resource requirements* and the classification quality for different parameter configurations of a *Support Vector Machine* (SVM) with a radial basis function kernel (RBF) that is commonly used in SVM classification tasks. Dependent on how the kernel parameter γ on the x-axis and the cost parameter C of constraint violations on the y-axis are chosen, different execution times and misclassification errors are produced. The left part of Fig. 6.1 shows the heat map for the execution times. Here, the runtime can vary heavily depending on the configuration. Dark blue configurations are long running configurations, while light green configurations have short execution times. The right part of Fig. 6.1 represents the heat map for the *mean misclassification error* (mmce). Here the performance criterion has the goal to find the configuration that has the lowest mmce (light yellow) within a limited time budget.

To reduce the number of necessary evaluations of the black-box function (e.g., a SVM classification like in Figure 6.1), conventional MBO uses an iteratively refined

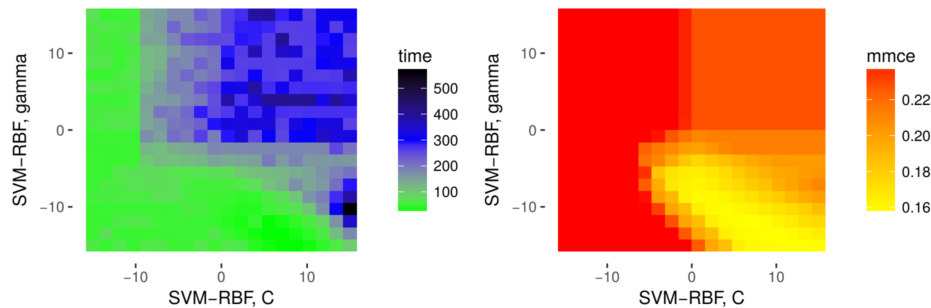

Figure 6.1: Runtime and mean misclassification error (mmce) for different configurations of an SVM classification.

regression model on a set of already evaluated configurations to approximate the objective function. Starting with an initial set of already evaluated configurations, the regression model guides the search to new promising configurations by estimating the outcome of the black-box function on yet unseen configurations. Based on this prediction, the so-called *infill criterion* (also called acquisition function) proposes a new promising configuration for evaluation. In each iteration, the regression model is updated based on evaluated configurations of all previous iterations until the budget is exhausted.

In its original formulation, the MBO algorithm operates purely sequentially (as described in Section 2.1), it proposes one configuration to be evaluated after the other [JSW98]. In order to propose multiple points simultaneously in a parallel setting, several modifications to the infill criteria or the general technique (such as constant liar, Kriging believer, qEI [GLC10], qLCB [HHL12], MOI-MBO [BWB+14]) have been suggested that result in multiple configurations being proposed in each iteration. The number of simultaneously proposed configurations is typically chosen to equal the number of available CPUs. However, these modifications in general neglect the heterogeneous resource requirements (CPU, memory etc.) for evaluating different configurations in the model space, which often leads to inefficient resource utilization.

Before new configurations are proposed, the results of all evaluations within one iteration are usually gathered to update the model. Thus the slowest evaluation becomes the bottleneck, and all other parallel worker processes idle after finishing their evaluation before a new MBO iteration can start. One approach to avoid idling is to desynchronize the model update. Such asynchronous techniques have been suggested and discussed by [GJL11; JLG11; JLG+12]. Here, the main problem is to avoid evaluations of very similar configurations. Since the evaluation of a configuration can take *several hours*, the evaluation of similar configurations with little or no impact on the overall optimization are a waste of resources.

The goal is to execute MBO in a resource-efficient way to enable the processing of larger problem sizes within a given time budget or, in other words, reduce the

end-to-end wall-clock time for a constant problem size. This calls for the development of new resource-aware scheduling strategies to efficiently map configurations to the underlying parallel architecture, depending on their resource demands. In contrast to classical scheduling problems, the scheduling for MBO needs to interact with the configuration proposal mechanism to select configurations with suitable resource demands for parallel evaluation, which is a complex problem, since the resource demands need to be known (at least estimated) before execution.

This chapter presents the *Resource-Aware Model-Based Optimization* framework (RAMBO), enabling a resource-efficient parallel variant of MBO by integrating different resource-aware scheduling strategies. With RAMBO, it becomes possible to make use of the full potential of parallel architectures in an efficient manner. To accomplish this goal a runtime estimation model that estimates the runtime for each evaluation of a configuration is developed to guide the mapping of evaluations to available resources. In addition to the runtime estimates, the scheduling strategies use an execution priority reflecting the estimated profit of an evaluation for finding the best configuration. Different experimental setups are used to evaluate the performance of the RAMBO framework including the new scheduling strategies.

This chapter is structured as follows: First, the fundamentals and the related approaches of parallel MBO are presented in Section 6.1. An overview of the RAMBO framework that includes the new resource-aware MBO method developed in this thesis is given in Section 6.2. Section 6.3 presents the new resource-aware scheduling strategies, including their evaluation and comparison to the existing parallel MBO variants on homogeneous multiprocessor cluster systems. Section 6.4, in contrast, proposes a concept for the new resource-aware scheduling strategies for MBO on heterogeneous embedded systems. Finally, the results are summarized in Section 6.5.

6.1 Parallel Execution of Model Based Optimization - Existing Approaches

For applications like hyperparameter tuning for machine learning algorithms or computer simulations, parallelization has become of increasing interest to reduce the overall execution time [HVC16]. The most important parallel extensions of MBO that this thesis focuses on update the regression model either synchronously or asynchronously. The *synchronous* variant uses infill criteria with multi-point proposals wherein each MBO iteration multiple configurations are proposed and evaluated in parallel. Here, the model is updated after all evaluations within one iteration are finished. The *asynchronous* variant updates the model every time an evaluation is finished. In this case, each worker process generates one new configuration proposal individually. Both variants are based on different infill criteria and have different challenges presented in the following.

6.1.1 Parallel Synchronous Execution

To allow for parallelization with a synchronous model update, infill criteria and techniques (constant liar, Kriging believer, qEI [GLC10], qLCB [HHL12], MOI-MBO [BWB+14]) have been suggested that propose multiple configurations in each iteration. *Multi-point proposals* are able to derive q configuration proposals $\boldsymbol{x}_1^*, \ldots, \boldsymbol{x}_q^*$ simultaneously instead of only proposing one configuration $\boldsymbol{x}^*$ from a *surrogate model*. As described in the fundamentals of Chapter 2, the surrogate model is used as a regression model, as it is comparably inexpensive to evaluate and therefore often used when function evaluations are very expensive [FSK08].

Hutter et al. [HHL12] introduced the qLCB criterion which is an extension of the single-point LCB (2.4) criterion using an exponentially distributed random variable to generate q different candidate proposals by drawing random values of $\lambda_j \sim \text{Exp}(\lambda)$ $(j = 1, ..., q)$ from the exponential distribution.

$$\text{qLCB}(\boldsymbol{x}, \lambda_j) = \hat{\mu}(\boldsymbol{x}) - \lambda_j \hat{s}(\boldsymbol{x}) \text{ with } \lambda_j \sim \text{Exp}(\lambda) \tag{6.1}$$

The λ variable guides the exploration-exploitation trade-off. Sampling multiple different λ_j thus might result in different "good" configurations by varying the impact of the standard deviation term.

Figure 6.2 presents two exemplary MBO iterations where qLCB is used, proposing two configurations per iteration for parallel execution. As described in Figure 2.3 of Chapter 2, the y (solid line) in the upper parts of the two MBO iteration figures denotes the output of the unknown black-box function f, while $\hat{\boldsymbol{y}}$ (dotted line in upper parts) denotes the outputs of the surrogate regression model $\hat{f}$ that tries to approximate the black-box function. Gray areas around $\hat{\boldsymbol{y}}$ represent the uncertainty of the surrogate model.

In the first iteration step, the initial set of configurations (red dots) are already evaluated. In the lower part of the visualization of iteration one, two dotted lines represent the qLCB infill criterion for $q = 2$ with two different values for λ with $\text{qLCB}(\boldsymbol{x}, \lambda_1)$ and $\text{qLCB}(\boldsymbol{x}, \lambda_2)$ to vary the impact of the standard deviation $\hat{s}(\boldsymbol{x})$. Based on these two different λ values, different minima for the infill criterion (lcb) are computed, to propose two new configurations (blue triangles).

In the second iteration, $\hat{f}$ is refitted with the evaluated configurations (green rectangles) and two new configurations are proposed (blue triangles). This process continues until the budget is exhausted. The qLCB criterion is comparably inexpensive for generating many independent candidate proposals and thus used in the resource-aware scheduling strategies for parallel MBO [RKB+16; KRL+17], which are part of this thesis.

Another popular multi-point infill criterion is the qEI criterion [GLC10] which directly optimizes the single-point EI (2.3) criterion over q points. As the computation of EI is using Monte Carlo sampling, it is quite expensive [CG13]. Therefore, a less expensive alternative, the *Kriging believer* approach [GLC10], is often chosen. Here, the first configuration is proposed based on the standard single-point EI criterion. Its

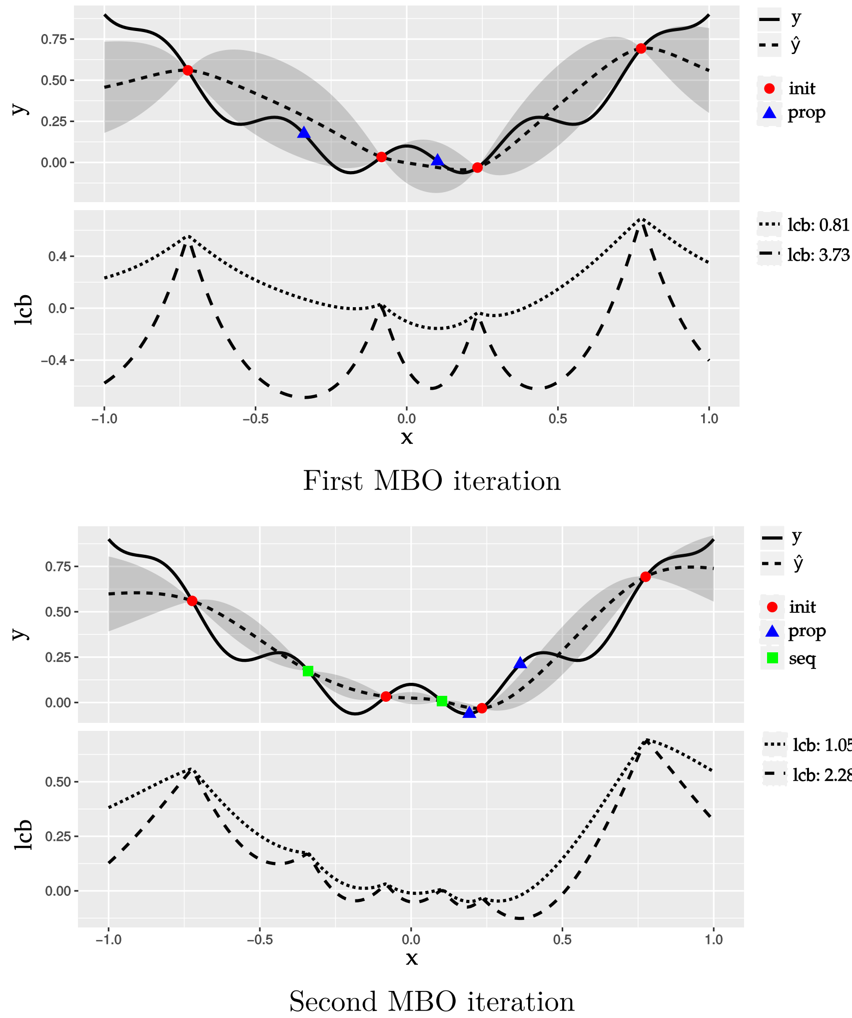

Figure 6.2: Visualization of two exemplary MBO iterations with qLCB as infill criterion and two parallel configuration evaluations per iteration.

posterior mean value is treated as a real value of f to refit the surrogate, penalizing the surrounding region with a lower standard deviation for the next point proposal using EI again. This is repeated until q proposals are generated.

The above mentioned multi-point infill criteria can cause inefficient resource utilization when the parallel executed evaluations have heterogeneous resource demands like execution times. For the synchronous parallel execution of MBO, the number q of proposed configurations is usually chosen to equal the number of available CPUs.

Figure 6.3 shows an exemplary schedule for $q = 4$ jobs (evaluations) on 4 CPUs where the jobs have varying execution times. The vertical arrows indicate the start of an MBO iteration where the multi-point criterion proposes 4 job configurations

$x_1, \ldots, x_4$ for the first iteration. The boxes represent the jobs execution times. At the end of the iteration, the model gets updated with the results of the configurations $x_1, \ldots, x_4$. All CPUs have to wait for the slowest evaluation (e.g., configuration x_2 in iteration one) to finish before receiving new proposals. This can lead to idling CPUs that are not contributing to the optimization. Here, spaces between jobs and the vertical arrow (model update) indicate idling CPU time caused by heterogeneous execution times of the jobs executed within one MBO iteration. After the results of all jobs are gathered the model is updated synchronously and new proposals can be generated and executed for the second MBO iteration. The general goal is to use the underlying parallel architecture in a resource-efficient way to solve the optimization problem.

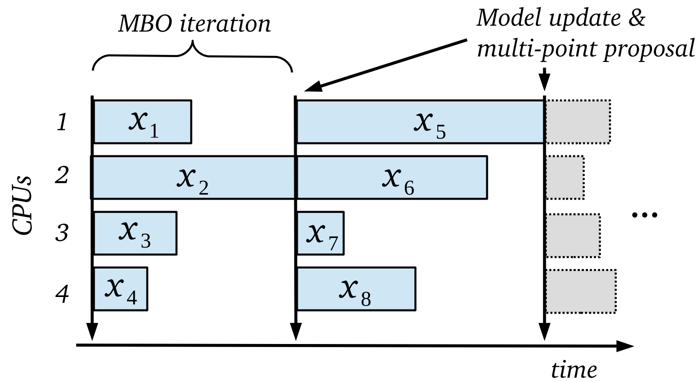

Figure 6.3: Exemplary scheduling for synchronous parallel MBO with $q = 4$ executed evaluations (jobs) per MBO iteration, with varying execution times leading to idling CPUs.

Varying execution times of parallel evaluations have already been addressed by Snoek et al. [SLA12] where the authors suggest to model these with an additional surrogate leading to an "expected improvement per second", favoring less expensive configurations. The parallel MBO approaches developed in this thesis also use regression models to estimate resource requirements, but instead of adapting the infill criterion, they use it to guide the scheduling of parallel evaluations. Here, the goal is to guide MBO to interesting regions in a faster and resource-efficient way without directly favoring less expensive configurations.

Another approach that addresses heterogeneous execution times of configuration evaluations executed in parallel is the asynchronous execution of MBO, which is described in the next paragraph.

6.1.2 Parallel Asynchronous Execution

In the asynchronous parallel execution approach, instead of evaluating multiple configurations in batches and synchronously refit the model, the model is refitted after each evaluation to avoid CPU idling. Here, the number of worker processes equals the

number of available CPUs, but each worker proposes the next point for evaluation independently, even if configurations $\boldsymbol{x}_{\text{busy}}$ are currently under evaluation on other CPUs. The main challenge is to avoid evaluations of very similar configurations by modifying the infill criterion to deal with points that are currently under evaluation. Here, the less expensive Kriging believer approach [GLC10] that is based on EI (also used for multi-point proposals) can be applied to block these regions.

Another approach that imputes pending values is the *Expected* EI (EEI) [GJL11; JLG+12; SLA12]. Here, the unknown value of $f(\boldsymbol{x}_{\text{busy}})$ is integrated out by calculating the expected value of y_{busy} via Monte Carlo sampling, which is, similar to qEI, computationally demanding. For each Monte Carlo iteration values $y_{1,\text{busy}}, \ldots, y_{\mu,\text{busy}}$ are drawn from the posterior distribution of the surrogate regression model at $\boldsymbol{x}_{1,\text{busy}}, \ldots, \boldsymbol{x}_{\mu,\text{busy}}$, with μ denoting the number of pending evaluations. These values are combined with the set of already known evaluations and used to fit the surrogate model. The EEI can then simply be calculated by averaging the individual expected improvement values that are formed by each Monte Carlo sample (n_{sim} denotes the number of Monte Carlo iterations):

$$\widehat{\text{EEI}(\boldsymbol{x})} = \frac{1}{n_{\text{sim}}} \sum_{i=1}^{n_{\text{sim}}} \text{EI}_i(\boldsymbol{x}) \tag{6.2}$$

Besides the advantage of an increased CPU utilization, asynchronous execution can also cause additional runtime overhead due to the higher number of model updates and the computational costs for new point proposals, especially when the number of available CPUs increases. Furthermore, heterogeneous execution times of job configurations can lead to very similar point proposals due to model updates that are based on similar histories.

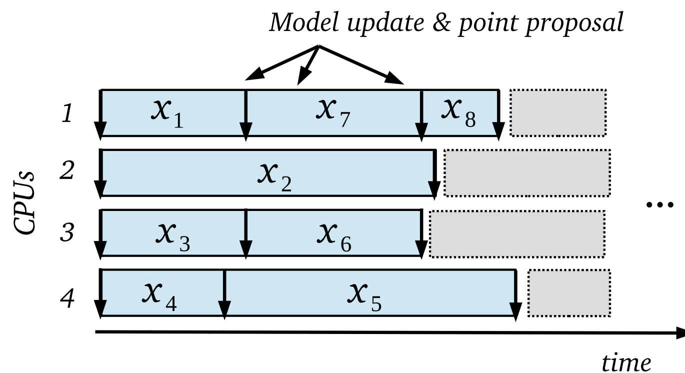

Figure 6.4: Exemplary scheduling for asynchronous parallel MBO to avoid CPU idling, with varying execution times that can lead to evaluations of similar configurations.

Figure 6.4 shows an exemplary schedule for asynchronous parallel MBO. Here, each worker process proposes the next configuration for evaluation (job) itself, indicated by the vertical arrows. Like in Figure 6.3 the boxes represent the execution

times of the jobs. New configurations like x_7 on CPU 1 are evaluated even if other configurations like $\boldsymbol{x}_{2,\text{busy}}$ on CPU 2 are still under evaluation. When a job finishes, its result is combined with the set of already known results and used to update the model and propose a new configuration.

This mechanism can cause evaluations of very similar configurations since the combined results may have the same history. For example, when the evaluations of x_1 on CPU 1 finishes (vertical arrow), the model update and the new proposal are based on the combination of results $f(x_1)$, $f(x_3)$ and $f(x_4)$. At the approximately same time, the model update performed after x_3 has the same history of results ($f(x_1)$, $f(x_3)$, $f(x_4)$) to perform the point proposal. Here, x_7 and x_6 could be very similar configurations that are evaluated in parallel and thus can lead to wasting resources. Furthermore, model updates that are based on the same history lead to computational overhead.

Instead of using asynchronous execution to efficiently utilize parallel computer architectures, the new approach developed in this thesis uses the synchronous execution combined with resource-aware scheduling. Section 6.3.4 will present a comparison of this approach with the above-described synchronous and asynchronous parallel variants of MBO.

6.2 Resource-Aware Model-Based Optimization

The synchronous and asynchronous parallel MBO variants described above neglect the resource requirements (CPU, memory etc.) for evaluating different configurations in parallel, which can lead to inefficient resource utilization.

The asynchronous parallel variant of MBO has the advantage of reducing the CPU-idle time by desynchronizing the model update. However, the frequent model updates after each evaluation can cause computational overhead for the configuration proposal. In addition, model updates that are based on similar histories of results can cause evaluations of very similar configuration. The evaluation of a configuration can take several hours, thus evaluating similar configurations is a waste of resources.

The synchronous parallel variant of MBO has the advantage of less frequent model updates and thus less computational overhead. Here the model is only updated per MBO iteration and multiple configurations are proposed simultaneously after each update. However, heterogeneous execution times of parallel evaluated configurations can cause CPU-idling and thus unused resources.

The goal of the new resource-aware MBO variant presented in this thesis is to reduce the end-to-end wall clock time needed for parallel MBO in a resource-efficient way and thus converge faster to the optimal configuration. To accomplish this goal this thesis proposes new resource-aware scheduling strategies to efficiently map parallel running evaluations to the underlying architecture, depending on their resource demands.

Resource-aware scheduling is an active field of research which is often tailored specifically for different hardware platforms, from small embedded systems [TLB+15] up to heterogeneous clusters [DK14; GAK+14]. In contrast to these classical scheduling problems, the new scheduling strategies for MBO are in control of the job generation and therefore interact with the job proposal mechanism (infill criterion) to postpone or skip suggested configurations if deemed not promising enough or if their resource demands are not suitable. Therefore, the resource demand of each configuration evaluation needs to be known or at least estimated before execution, which is a complex problem.

To enable the interaction between the new resource-aware scheduling strategies and the general MBO process, a new framework is proposed and presented in the next section.

RAMBO: Resource-Aware Model-Based Optimization Framework

The optimization cycle of the Resource-Aware Model-Based Optimization framework (RAMBO) is shown in Figure 6.5. RAMBO aims at efficient utilization of parallel computer architectures through resource-aware scheduling strategies to guide MBO to interesting regions of the black-box function in a faster and resource efficient way. Its foundations are based on the `mlrMBO` library [BBH+14].

RAMBO consists of three main steps. In the first step, the *MBO Method* builds the surrogate model according to the MBO algorithm and settings based on previous evaluations. At the same time, the *Job Utility Estimator* produces job profiles for the configurations through an additional regression model. These job profiles include runtime estimates that are later used as input for the scheduling strategies.

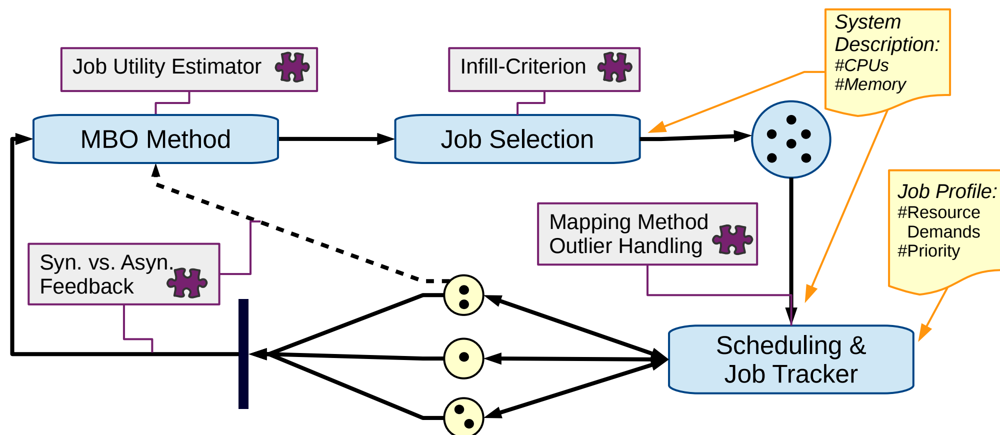

Figure 6.5: Resource-Aware Model-Based Optimization Framework.

The *Job Selection* follows the MBO principles for point proposals and generates a set of candidate proposals based on the selected infill criterion and the available resources (System Description). In combination with the estimated resource demands of a proposed point (derived from the regression model) a job is formed. For each job,

a priority is calculated based on the infill criterion. Finally, the *Scheduling* strategy allocates the jobs (configurations to evaluate) based on the system description and based on the job profile to the available resources.

Job profiles for the proposed configurations are only estimated, thus under- or overestimation, e.g., of execution times, can occur. In such a case, a job might need to be rescheduled or stopped to guarantee efficient resource utilization by the *Job Tracker* that monitors the execution. After a job has finished, there are two possibilities to update the model. For the *Synchronous Feedback* the results of all jobs within one MBO iteration are gathered before the model is updated following the synchronous parallel execution. For the *Asynchronous Feedback* each result directly triggers a call of the MBO method to update the model. Whether asynchronous or synchronous feedback is chosen also depends on the costs of a model update.

Since RAMBO can be configured with different MBO methods, infill criteria, and scheduling strategies, it not only includes new resource-aware MBO scheduling methods but also supports the development and evaluation of new MBO methods. The new RAMBO scheduling strategies including the job utility estimation and the job priorities will be presented and evaluated in the following.

6.3 Resource-Aware Scheduling Strategies

The scheduling strategies presented in this thesis are aiming at guiding MBO to interesting regions in a faster and resource-aware way. The goal is to acquire the feedback of the workers in the shortest possible time to avoid MBO model update delay while reducing the CPU idle time on the workers. To efficiently map the proposed jobs to the available resources, the scheduling needs to know the resource demands of each job before execution. Additionally, an execution priority is needed to update the model with the most promising jobs as soon as possible. Both the estimated resource utilization and the priority of the proposed candidates are used as inputs for the scheduling strategies.

Priorities for Job Selection

To model the usefulness of a candidate for the objective function, Kriging is used as a surrogate regression model and qLCB (6.1) is used as multi-point infill criterion to generate a set of job proposals. Compared to the multi-point proposal qEI [GLC10], the qLCB criterion is more suitable since it is able to propose a set of independent candidates. qLCB can simultaneously generate q candidates by drawing q random values of $\lambda_j \sim \text{Exp}(\lambda)$ $(j = 1, \ldots, q)$ from the exponential distribution. Each λ_j results in a different trade-off between exploitation ($\lambda_j \downarrow$) and exploration ($\lambda_j \uparrow$) and thus leads to a different optimal configuration $\boldsymbol{x}_j^*$ after solving

$$\boldsymbol{x}_j^* := \underset{\boldsymbol{x}}{\operatorname{argmin}} \left[\text{LCB}(\boldsymbol{x}, \lambda_j) \right] = \underset{\boldsymbol{x}}{\operatorname{argmin}} \left[\hat{y}(\boldsymbol{x}) - \lambda_j \hat{s}(\boldsymbol{x}) \right], \tag{6.3}$$

where $\hat{y}(\boldsymbol{x})$ denotes the posterior mean and $\hat{s}(\boldsymbol{x})$ denotes the root of the posterior standard deviation of the surrogate model at point $\boldsymbol{x}$.

Since there is no direct order of the set of proposed candidates $\boldsymbol{x}_j^*$ in terms of how promising one candidate is, an additional order is introduced to guide the search for the best candidate towards more promising areas. Therefore, the highest priority is given to the point $\boldsymbol{x}_j$ that was proposed using the smallest value of λ_j and is thus closest to the optimum (exploitation). The priority for each job is defined as $p_j := -\lambda_j$.

Resource Utility Estimation

The runtime estimates of the set of jobs proposed in each MBO iteration are needed for the scheduling to avoid execution of jobs with high runtime variances and thus reduce idling worker processes. This is accomplished by using an additional regression model. In the same way, as for the MBO algorithm itself, the runtime of a job is predicted in each iteration based on the runtime of all previously evaluated jobs to build the runtime model of the black-box function. For the model, Kriging is used for homogeneous CPU systems, since the runtime is expected to be a continuous function. For parallel architectures with heterogeneous CPUs, Random Forest is used for the model instead. Here, the runtime of a job is estimated for different CPU types (as described in Section 6.4).

The accuracy of the runtime estimation also influences the scheduling decision. Thus a scheduling strategy is only able to efficiently map jobs to the underlying parallel architecture if the estimates are reliable. Therefore the runtime estimation quality is evaluated in Section 6.3.4. The following section will present two different scheduling strategies that are included in the RAMBO framework.

6.3.1 First Fit Scheduling Strategy

One of the first scheduling strategies included in the RAMBO Framework is a First Fit heuristic [RKB+16], which takes the parallel synchronous execution of MBO as a basis. The goal is to reduce the CPU idle time on the workers that evaluate the configurations while acquiring the results in the shortest possible time to avoid model update delay. The set of candidates proposed by the multi-point infill criterion qLCB forms a set of jobs $J = \{1, \dots, q\}$ that should be executed on the available CPUs $K = \{1, \dots, m\}$ within each MBO iteration. For each job the estimated runtime is given by $\hat{t}_j$ and the corresponding priority is given by p_j.

To reduce idle times caused by evaluations of configurations with a low priority, the maximal runtime for each MBO iteration is defined by the runtime of the job with the highest priority $j^* := \arg\max_j p_j$. Lower prioritized jobs have to subordinate. In this way, the surrogate model of the black-box function is updated as soon as possible with the results of the most promising configuration (highest priority indicated by the infill criterion) which optimizes the job proposal for the next MBO iteration and thus helps the search to progress quickly.

In each MBO iteration, a list of $q = 3m$ job proposals is generated, where m is the number of available CPUs. Note that for a useful scheduling, the number of candidates q should be considerably larger than available CPUs. Then the job with the highest priority is determined and mapped to CPU $k = 1$ exclusively. Accordingly, on a system with m homogeneous CPUs the remaining jobs are scheduled on CPUs $2, \ldots, m$, limited by the upper time bound $\hat{t}_{j^*}$, which is directly derived from the estimated runtime of job j^*. Jobs which have an estimated runtime $\hat{t}_j \leq \hat{t}_{j^*}$ are mapped in decreasing order of their priorities to the remaining CPUs in a greedy first fit manner. A job j is mapped onto CPU k if its runtime satisfies $\hat{t}_j \leq \hat{t}_{j^*} - \sum_{j \in J_k} \hat{t}_j$ where J_k is the set of jobs already mapped to CPU k. Jobs included in the initial list J that do not fit on any CPU are discarded and will be proposed again in the next MBO iteration if they are promising enough.

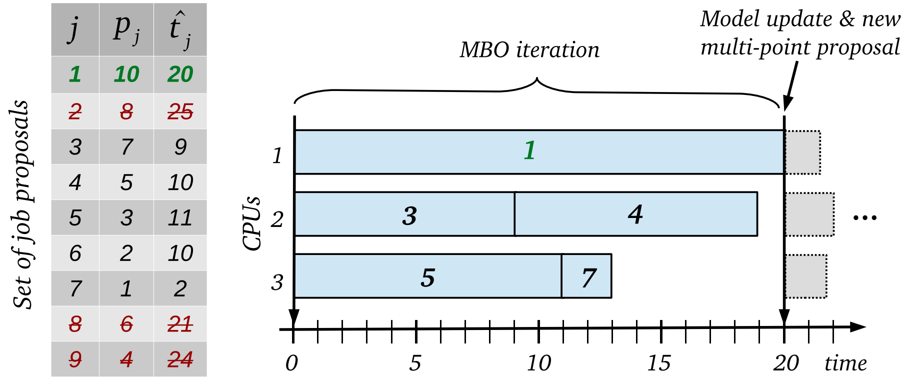

Figure 6.6: Exemplary schedule of the resource-aware First Fit strategy with $q = 9$ jobs and $m = 3$ CPUs.

Figure 6.6 shows an exemplary schedule for the resource-aware First Fit scheduling strategy. Here, the set of job proposals consists of $3m = 9$ jobs with $m = 3$ CPUs (see left table in Figure 6.6). The job with the highest priority $j = 1$ (marked in green) is mapped to CPU $k = 1$ exclusively. This job defines the time bound $\hat{t}_{1^*} = 20$ for the MBO iteration. The jobs that are marked in red do not fit this time bound and are thus discarded $(j = 2, j = 8, j = 9)$. The remaining jobs are mapped with the First Fit heuristic, in decreasing order of their priorities p_j on CPU 2 and 3 if their runtime $\hat{t}_j$ is smaller or equal to $\hat{t}_1 - \sum_{j \in J_k} \hat{t}_j$.

If any CPU is left without a job, the regression model for the runtime estimation can be optionally queried for new jobs with a runtime smaller or equal to $\hat{t}_{j^*}$ to fill gaps. When all scheduled jobs are evaluated, both regression models – one for the black-box function and one for the runtime estimates of the black-box function – are updated and the MBO iteration starts over.

Evaluation of the First Fit Scheduling Strategy

To evaluate the First Fit scheduling strategy for MBO, it is compared to two established alternatives, namely the random search [BB12] and the synchronous qLCB approach [BWB+14].

RS: The random search is a naive but often effective approach that asynchronously evaluates a randomly selected point directly after each evaluation to guarantee maximum CPU load.

qLCB: The synchronous qLCB approach uses the multi-point qLCB criterion, with Kriging as a surrogate model. In each MBO iteration as many configurations as available CPUs $q = m$ are proposed and evaluated to solve with random $\lambda_j \sim \text{Exp}(\frac{1}{2})$.

As benchmark for the comparison, a hyperparameter optimization of a SVM with RBF kernel is used.

$$k(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \left\| \mathbf{x} - \mathbf{x}' \right\|^2). \tag{6.4}$$

The kernel parameter γ and the cost C of constraint violations are both box-constrained to the interval $[-15, 15]$ on a $\log_2$-scale. The SVM implementation is based on `libsvm`[1] and implemented in the R library `e1071` [ADK06]. Two data sets `w6a`[2] and `magic04`[3] are used as input for the SVM kernel parameter optimization.

To interface the SVM machine learning algorithm, the R library `mlr` [BLK+16] is used. The synchronous qLCB approach and the random search are implemented in the R library `mlrMBO` [BBH+14], which is also the basis for the RAMBO framework that includes the First Fit strategy. To parallelize the evaluation on high performance clusters, the `BatchExperiments` [BLM+15] R library is used. For the evaluation of the results, an outer 10-fold cross-validation is used while a 3-fold cross validation is used to define the objective function for the hyperparameter tuning. For comparability reasons, all optimization approaches start with the same initial latin hypercube design [MBC00] including 10 evaluated configurations and are conducted on $m = 4$ CPUs. The initial designs are fixed per outer cross-validation fold. The budget for each approach is defined via the elapsed time.

For the comparison the mean misclassification error (MMCE) of the best SVM configuration found after 1, 10, 120 and 180 minutes across all 10 cross-folds is shown in Figure 6.7 (lower is better). Both input data sets `w6a` and `magic04` are separated into test and training data sets for the hyperparameter tuning of the SVM. The left hand side of Figure 6.7 shows the MMCE of the hyperparameter tuning data sets, while the right hand side represents the MMCE of the test data sets for both `w6a` and `magic04` (lower is better). On these data sets only marginal improvements are achieved with the RAMBO First Fit strategy. Still, RAMBO yields comparable

[1] Chang, C. et. al: https://www.csie.ntu.edu.tw/~cjlin/libsvm

[2] Platt: http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/w6a

[3] Bock: https://archive.ics.uci.edu/ml/datasets/MAGIC+Gamma+Telescope

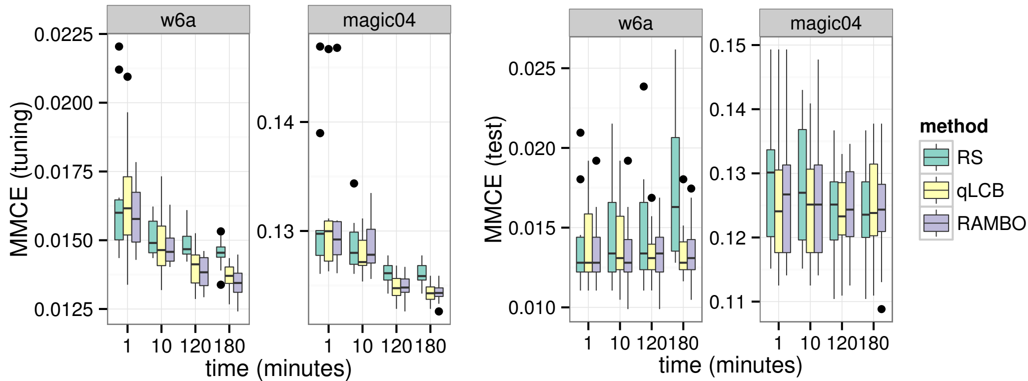

Figure 6.7: Averaged mean misclassification errors (MMCE): tuning error (left) and test data error (right) for the best observed configuration within a given time budget (lower is better).

performance and sometimes less variance compared to the synchronous approach qLCB. A low variance across all 10 cross-folds is important since it indicates a higher confidence in the optimization result.

The reason for less variance in the RAMBO results lies in the optimized scheduling. Figure 6.8 presents an exemplary visualization for the scheduling of the parallel executed SVM configurations (jobs) for qLCB and RAMBOs First Fit strategy. Each gray box represents the execution of a job on the respective CPU (see y-axis).

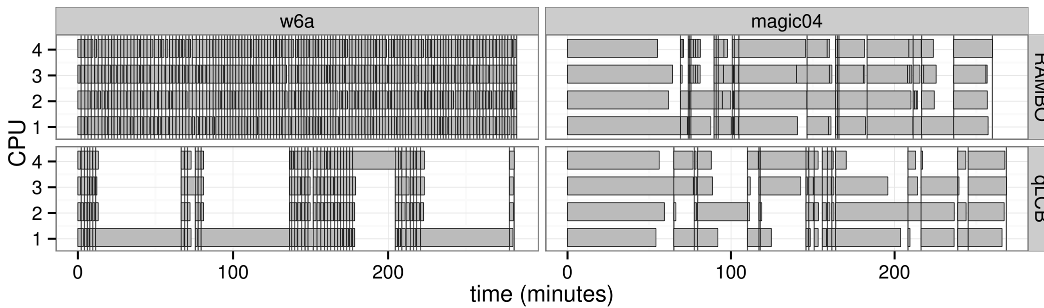

Figure 6.8: Scheduling of MBO approaches: Runtime on x-axis and mapping of proposed candidates (gray boxes) to $m = 4$ CPUs on y-axis. Vertical lines indicate the end of a MBO iteration. Gaps represent idle time.

The vertical lines indicate the end of a MBO iteration. Here, for both `w6a` (left) and `magic04` (right) the necessity of the resource estimation for jobs with heterogeneous execution times becomes obvious, as qLCB (lower part of 6.8) can cause long idle times by running jobs with large runtime differences together within one MBO iteration. In contrast, the First Fit strategy of RAMBO balances long execution times more evenly (upper part of 6.8). The runtime estimation is reliable so that only 2.3% of the evaluations exceed $\hat{t} + 2 \cdot s(\hat{t})$. The visualization of the scheduling on `magic04` also shows that the resource-aware First Fit Strategy not only prefers short jobs but is also able to schedule long jobs more efficiently. On

w6a twice as many SVM configurations are evaluated compared to the synchronous qLCB approach in the given time budget. In contrast, 25% more configurations were evaluated on magic04 indicating that promising configurations have longer runtimes than average and vice versa for w6a.

Those first results show that the First Fit scheduling heuristic of RAMBO already leads to improved resource utilization and thus to more evaluations within the same time budget potentially yielding a better knowledge of the hyperparameter space. Besides the above-described First Fit scheduling strategy, an additional optimized version based on the knapsack algorithm is also integrated into the RAMBO framework [KRL+17] and will be presented in the next section.

6.3.2 Knapsack based Scheduling Strategy

As for the First Fit heuristic, the goal of the knapsack based scheduling strategy is also to reduce the CPU idle time on the workers while acquiring the feedback of the workers in the shortest possible time to avoid model update delay. Here the qLCB multi-point infill criterion is used to form a set of jobs $J = \{1, \ldots, q\}$ that should be executed on the available CPUs $K = \{1, \ldots, m\}$. As for the First Fit strategy the estimated runtime is given by $\hat{t}_j$ and the corresponding priority by p_j for each job proposal. The time bound for each MBO iteration is here also defined by the runtime of the highest prioritized job.

The goal is to maximize the profit, given by the priorities, of parallel job executions within each MBO iteration. To solve this problem, we apply the $0-1$ multiple knapsack algorithm for global optimization routines [Bor16]. Here, the knapsacks are the available CPUs and their capacity is the maximally allowed computing time, defined by the runtime of the job with the highest priority. The items are the jobs J, their weights are the estimated runtimes $\hat{t}_j$ and their values are the priorities p_j. The capacity for each CPU is accordingly $\hat{t}_{j^*}$, with $j^* := \arg\max_j p_j$. To select the best subset of jobs the algorithm maximizes the profit Q:

$$Q = \sum_{j \in J} \sum_{k \in K} p_j c_{kj}, \tag{6.5}$$

which is the sum of priorities of the selected jobs, under the restriction of the capacity

$$\hat{t}_{j^*} \geq \sum_{j \in J} \hat{t}_j c_{kj} \ \forall k \in K \tag{6.6}$$

per CPU. The restriction with the decision variable $c_{kj} \in \{0, 1\}$

$$1 \geq \sum_{k \in K} c_{kj} \ \forall j \in J, c_{kj} \in \{0, 1\}. \tag{6.7}$$

ensures that a job j is at most mapped to one CPU.

As the job with the highest priority defines the time bound $\hat{t}_{j^*}$ it is mapped to the first CPU $k = 1$ exclusively and single jobs with higher execution times are

directly discarded. (Discarded jobs will be proposed again in the next MBO iteration if they are promising enough.) Then the knapsack algorithm is applied to assign the remaining candidates in J to the remaining $m-1$ CPUs. This leads to the best subset of J that can be run in parallel minimizing the delay of the model update. As for the First Fit heuristic, if a CPU is left without a job the regression model can be optionally queried for a job with an estimated runtime smaller or equal to $\hat{t}_{j^*}$ to fill the gaps.

Figure 6.9 shows an exemplary schedule for the above-described resource-aware knapsack based scheduling strategy. Here, the set of job proposals $q = 9$ with 3

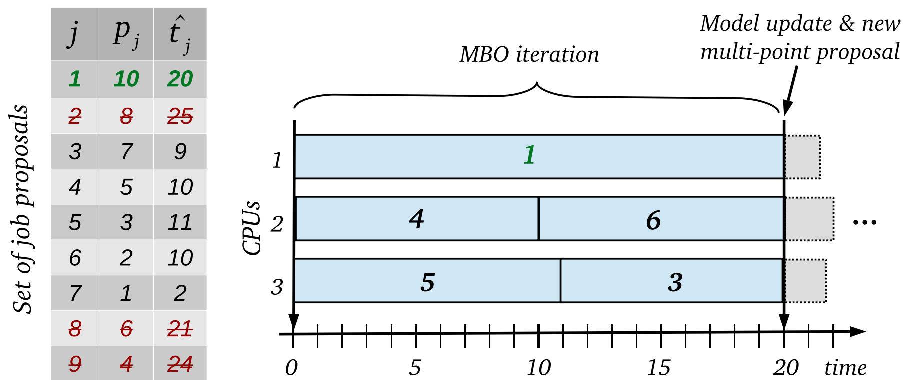

Figure 6.9: Exemplary schedule of the resource-aware knapsack based strategy with $q = 9$ jobs and $m = 3$ CPUs.

CPUs is similar to the First Fit strategy example in Figure 6.6. The job $j = 1$ with the highest priority $p_1 = 10$ is mapped to CPU $k = 1$ exclusively. The time bound for the MBO iteration and thus the capacity for each CPU is given by the runtime of job $j = 1$. Jobs that do not fit this time bound are discarded ($j = 2, j = 8, j = 9$). To maximize the profit, given by the priorities the knapsack algorithm is applied to map the remaining jobs to CPU $k = 2$ and $k = 3$. In comparison to Figure 6.6 that illustrates the First Fit strategy, the jobs are mapped more evenly and thus no CPU idle time occurs.

To reduce computational overhead the First Fit heuristic is integrated into the knapsack implementation and is applied instead of the knapsack algorithm if the number of remaining jobs in J is equal to the number of available CPUs. Before evaluating the knapsack based scheduling approach a refinement of the job priorities is presented in the next section, which can be optionally applied in combination with the proposed scheduling strategies.

6.3.3 Refinement of Job Priorities

The refinement of job priorities has the goal to avoid parallel evaluations of very similar configurations and can be optionally used for all scheduling strategies included in the RAMBO framework. Approaches to specifically propose configurations that

are promising but yet diverse are described in [BWB+14]. qLCB performed well and was chosen here because it is comparably inexpensive to create many independent candidates. However, qLCB does not include a penalty for the proximity of selected configurations, which might become a problem if the number of parallel evaluations is high. Therefore, the Euclidean distance is used to reprioritize p_j to $\tilde{p}_j$, encouraging the selection of configurations that are more scattered in the domain space.

First, a set of $q > m$ configurations is sampled from the qLCB criterion. These configurations are then hierarchically clustered by their distance in the domain space of the objective function using the complete linkage method. The procedure starts with the configuration that has previously been assigned the highest priority and assigns it to the first position in the list of selected jobs $\tilde{J}$. For each following step $i \geq 2$, all candidates are split into i clusters according to the hierarchical clustering. Of these i clusters the $i - 1$ clusters that already contain candidates with assigned positions are discarded, leaving one cluster. The position i in $\tilde{J}$ is assigned to the job with the highest priority within this cluster. This is continued until all q candidates have assigned positions. Thereby an ordering that follows the hierarchy induced by the clustering is generated. Finally, new priorities $\tilde{p}_j$ are assigned based on the order of $\tilde{J}$, i.e. the first job in $\tilde{J}$ gets the highest priority q and the last job gets the lowest priority 1.

As a result, the set of candidates contains batches of jobs with similar priority that are spread in the domain space. The new priorities serve as input for the scheduling which groups the q jobs to m CPUs using the runtime estimates $\hat{t}$.

6.3.4 Evaluation

To evaluate the resource-aware MBO scheduling strategies that are included in the RAMBO framework a comparison with different synchronous and asynchronous parallel MBO approaches is performed. The comparison includes two asynchronously executed MBO strategies that aim at using all available CPU time to solve the optimization problem in parallel. Both of them [GJL11; JLG11] use Kriging as a surrogate, with the EEI criterion (6.2) [JLG+12] and the Kriging believer [GLC10] criterion. In Kotthaus et al. [KRL+17] RAMBO was also compared to a third asynchronous execution strategy that is included in the SMAC (Sequential Model-based Algorithm Configuration) tool [HHL11] which uses a Random Forest surrogate. The results showed that RAMBO and the two other asynchronous execution strategies always converged faster to the optimum compared to SMAC, which is why SMAC is not included in the following evaluation.

Besides the comparison with the asynchronous strategies, the following evaluation also includes two synchronously executed MBO approaches. One of them uses the qLCB multi-point infill criterion (6.1) and one uses the qEI criterion [GLC10]. All parallel MBO approaches including the new RAMBO approach are evaluated on a set of established continuous synthetic functions combined with simulated execution times to ensure a fair and disturbance-free environment.

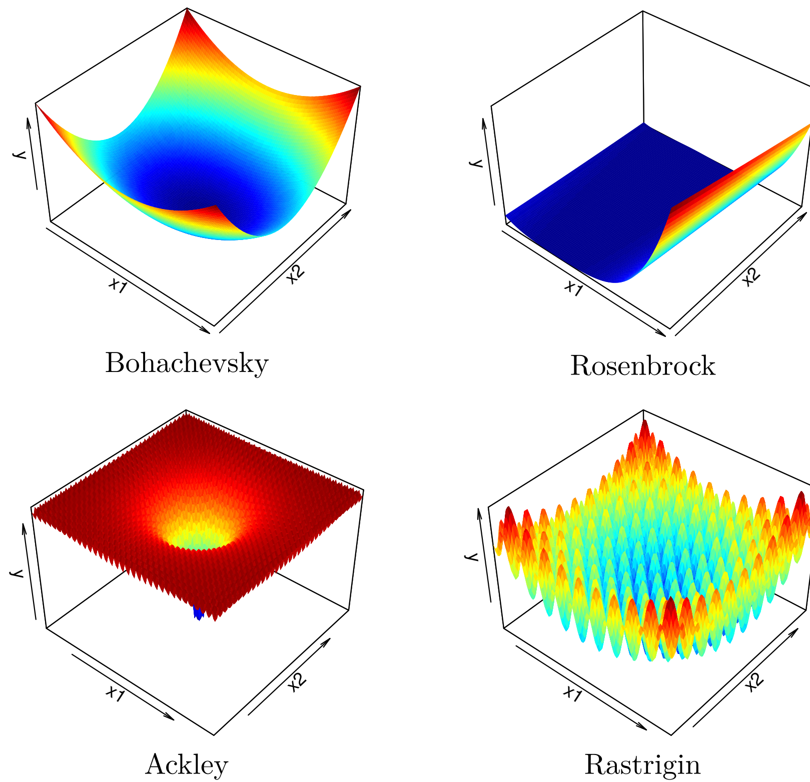

Figure 6.10: Visualization of the synthetic test functions for $d = 2$, used for the evaluation. `bohachevsky`(d) and `rosenbrock`(d) (upper part) show a smooth surface while `ackley`(d) and `rastrigin`(d) (lower part) are highly multimodal.

The usage of synthetic functions rules out technical problems emerging on multi-user systems (swapping, network congestion, CPU cycle stealing, other users occupying fast caches, . . .). Furthermore, synthetic functions ease the evaluation of MBO approaches on different difficulty levels. Therefore two different categories of objective functions (implemented in the `R` library `smoof` [Bos16]) are considered:

1. Functions with a smooth surface: `rosenbrock`(d) and `bohachevsky`(d) with dimension $d = 2, 5, 10$, that are likely to be fitted well by MBO.
2. Highly multimodal functions: `ackley`(d) and `rastrigin`(d) ($d = 2, 5, 10$), for which MBO is expected to have problems achieving good results.

For each objective function a 2-, 5- and a 10-dimensional version are used, each of which is optimized using 4 and 16 CPUs in parallel to investigate scalability. Figure 6.10 visualizes the synthetic test functions for $d = 2$.

Since synthetic functions are illustrative test functions, they have no significant runtime. Therefore, these functions are also used to simulate different runtime behaviors. For each benchmark two different synthetic functions are combined: One

determines the number of seconds it would take to calculate the objective value of the other function. E.g., for the combination `rastrigin(2).rosenbrock(2)` it would require `rosenbrock(2)(`$\boldsymbol{x}$`)` seconds to retrieve the objective value `rastrigin(2)(`$\boldsymbol{x}$`)` for an arbitrary proposed configuration $\boldsymbol{x}$. Technically, the benchmark sleeps `rosenbrock(2)(`$\boldsymbol{x}$`)` seconds before returning the objective value. The runtime is simulated with either `rosenbrock(`d`)` or `rastrigin(`d`)` and all combinations of the four objective functions are analyzed, except where the objective and the time function are identical. For the unification of the input space, values from the input space of the objective function are mapped to the input space of the function that simulated the runtime behavior. The output of the runtime functions are scaled to return values between 5 min to 60 min.

To examine how fast the parallel approaches converge to the optima of the benchmark functions within a limited time budget the distance between the best found configuration at time t and a predefined target value (optimal configuration) is measured. This measurement reflects the accuracy of the receptive MBO approach within the given time budget. To make this measurement comparable for all objective functions, the function values are scaled to $[0, 1]$. Here, 0 is the target value, defined as the best configuration y reached by any optimization approach within the given time budget. The upper bound 1 is the best y found in the initial set of already evaluated configurations and is identical for all approaches per given benchmark. Both values are averaged over 10 repetitions. If an optimization needs 2 h to reach an accuracy of 0.5, this means that within 2 h half of the way to the best configuration 0 has been accomplished, after starting at 1. The differences between the approaches are compared at three accuracy levels 0.5, 0.1 and 0.01. The optimizations are repeated 10 times and conducted on $m = 4$ and $m = 16$ CPUs to examine the scalability.

As time budget the optimization approaches run for 4 h on 4 CPUs and for 2 h on 16 CPUs in total including all computational overhead and CPU idling. All experiments are executed on a Docker Swarm cluster using the `R` library `batchtools` [LBS17]. The initial set is generated by latin hypercube sampling [MBC00] with $n = 4 \cdot d$ configurations and all of the following optimizations start with the same initial set in all 10 repetitions:

- **`rs`:** Random search, serves as a base-line.
- **`qLCB`:** Synchronously executed MBO approach using qLCB where in each MBO iteration $q = m$ configurations are proposed.
- **`ei.bel`:** Synchronously executed approach using Kriging believer where in each MBO iteration m configuration are proposed.
- **`asyn.ei.bel`:** Asynchronously executed MBO approach using Kriging believer.
- **`asyn.eei`:** Asynchronously executed MBO approach using EEI (100 Monte Carlo iterations).
- **`rambo`:** New synchronously executed MBO approach using qLCB with job priority refinement and the knapsack based resource-aware

scheduling strategy, where in each iteration $q = 8 \cdot m$ candidates are proposed.

qLCB and ei.bel are implemented in the R library mlrMBO [BBH+14]. asyn.eei, asyn.ei.bel and rambo are also based on mlrMBO. For all MBO approaches a Kriging model is used from the library DiceKriging [RGD12] with a Matern$\frac{5}{2}$-kernel [Mat60] and a nugget effect of $10^{-8} \cdot \mathrm{Var}(\boldsymbol{y})$, where $\boldsymbol{y}$ denotes the vector of all observed function outcomes.

For the evaluation, the rambo configuration with the knapsack based scheduling including the job priority refinement (as described in Subsection 6.3.3) is chosen since it delivered the best results compared to the other scheduling strategies.

Figure 6.11 shows an exemplary result comparing the scheduling strategies included in the RAMBO framework for rastrigin(10).rosenbrock(10) on 16 CPUs, which is one of the most complex benchmarks.

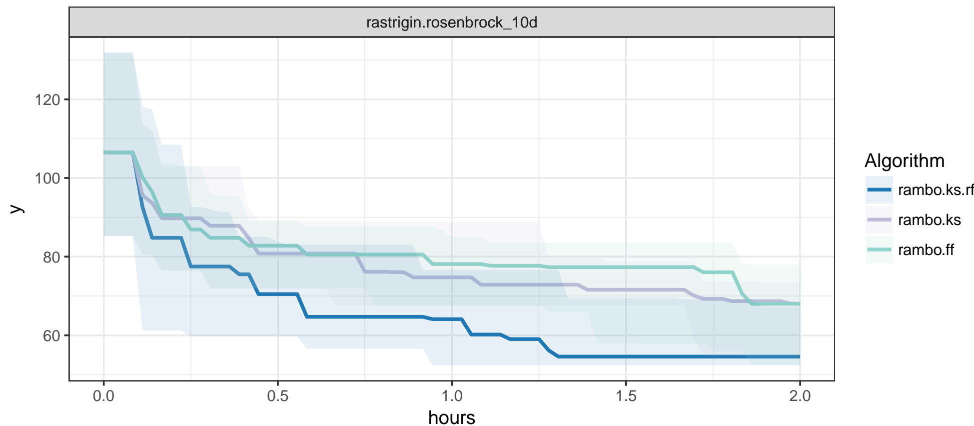

Figure 6.11: Averaged y value representing the best found configuration within a time budget of 2 h comparing the resource-aware scheduling strategies for the rastrigin(10) objective function with the rosenbrock(10) time functions on 16 CPUs (lower is better).

The y-axis represents the original (unscaled) y value of the configuration that was found after a certain amount of time, here, lower is better (closer to the optimal configuration). Transparent areas represent the deviations in the 10 repetitions while straight lines represent the median of these repetitions. The x-axis shows the runtime.

Here, the knapsack based strategy with the job priority refinement rambo.ks.rf outperforms its counterpart rambo.ks without the refinement as well as the First Fit heuristic rambo.ff that was presented in 6.3.1 and is thus selected in the following evaluation of the RAMBO framework.

Evaluation of the Resource Estimation

The quality of resource-aware scheduling naturally depends on the accuracy of the resource estimation. Without reliable runtime predictions, the scheduler is unable to optimize for efficient utilization. The runtime for all benchmarks is simulated with either `rosenbrock`(d) or `rastrigin`(d).

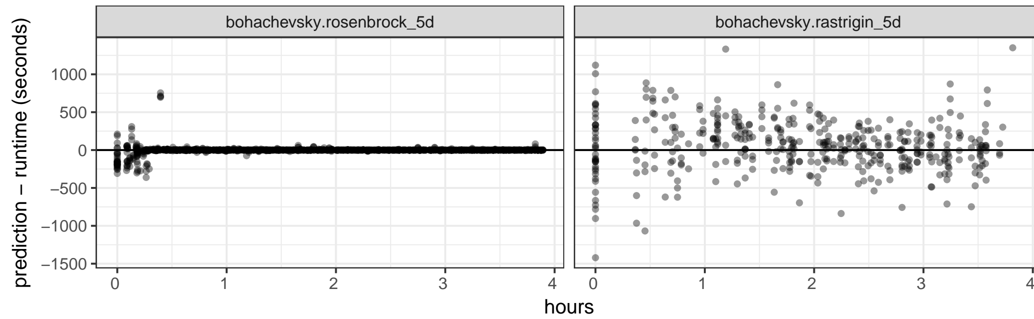

Figure 6.12: Residuals of the runtime estimation in the course of time for the `rosenbrock`(5) and `rastrigin`(5) time functions on 4 CPUs combined with `bohachevsky`(5) as objective function. Positive values indicate an overestimated runtime and negative values an underestimation.

Figure 6.12 exemplarily shows that the runtime estimation for the `rosenbrock`(5) time function works well (left part). Here, the residual values for the runtime estimation of the evaluated configurations are getting smaller over time. However, the runtime prediction for `rastrigin`(5) (right part) is comparably imprecise. For the 2 and 10-dimensional versions the results are similar.

This encourages to consider scenarios separately where runtime estimation has a high quality (`rosenbrock`($\cdot$)) and where runtime estimation is error-prone (`rastrigin`($\cdot$)), for further analysis.

Evaluation with High Resource Estimation Quality

Figure 6.13 shows box plots for the time required to reach the three different accuracy levels in 10 repetitions within a budget of 4 h on 4 CPUs (upper part) and 2 h on 16 CPUs (lower part). The faster an approach reaches the desired accuracy level, the lower the displayed box and the better the approach. If an approach was not able to reach an accuracy level within the given time budget, the respective time budget (4 h or 2 h) plus a penalty of 1000 s is inputted.

Table 6.1 lists the aggregated ranks over all objective functions, grouped by approach, accuracy level, and number of CPUs. For this computation, the approaches are ranked w.r.t. their performance for each repetition and benchmark before they are aggregated with the mean. If there are ties in Figure 6.13 (e.g., if an accuracy level was not reached), all values obtain the worst possible rank.

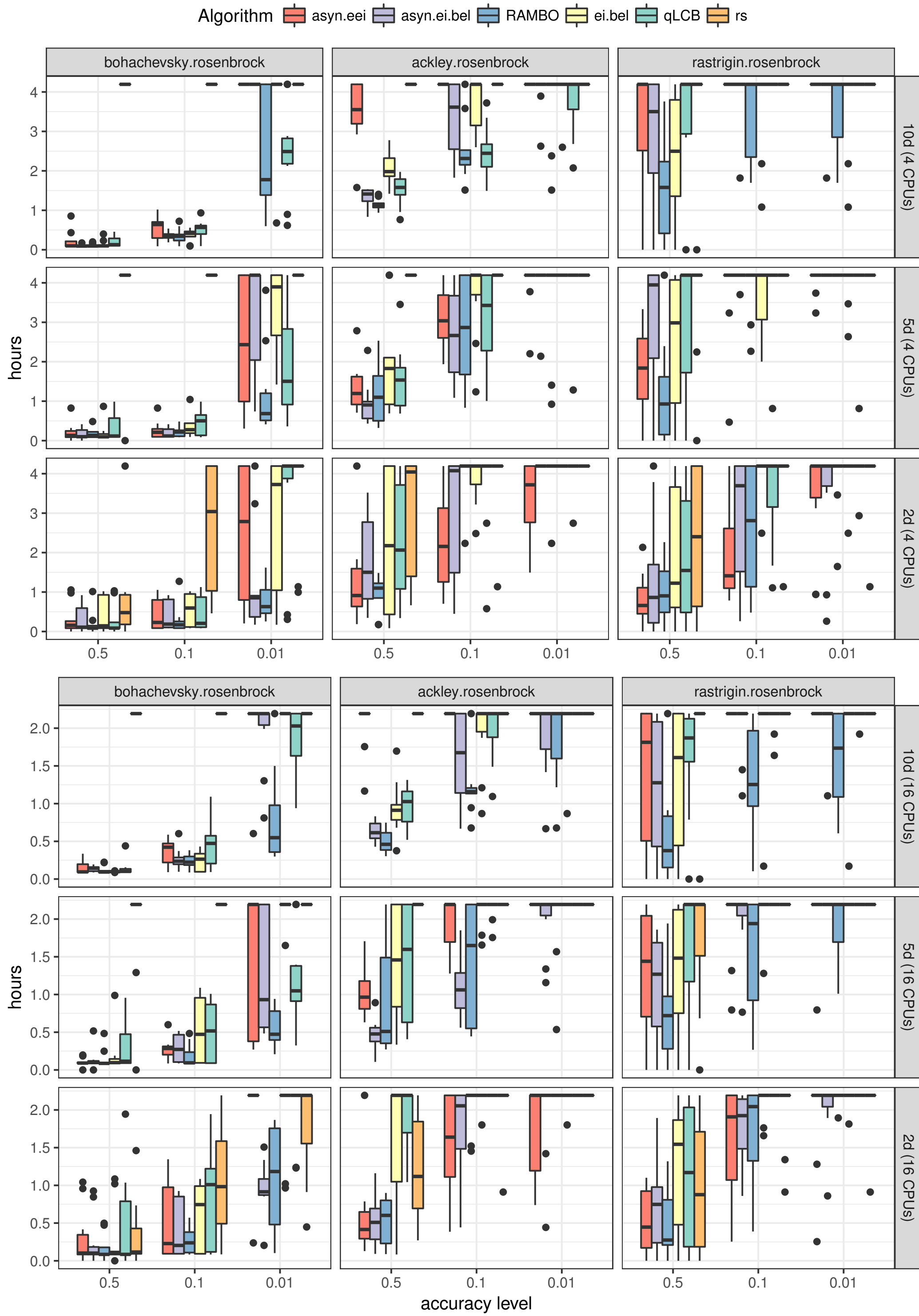

Figure 6.13: Accuracy level vs. execution time for different objective functions using time function `rosenbrock(·)` (lower is better).

	4 CPUs			16 CPUs		
Algorithm	0.5	0.1	0.01	0.5	0.1	0.01
`asyn.eei`	3.53 (3)	3.91 (3)	4.91 (3)	3.64 (3)	4.30 (3)	5.30 (3)
`asyn.ei.bel`	3.21 (2)	3.66 (2)	5.04 (4)	2.93 (2)	3.31 (2)	4.48 (2)
`rambo`	**2.47 (1)**	**3.40 (1)**	**4.23 (1)**	**2.54 (1)**	**2.98 (1)**	**3.72 (1)**
`ei.bel`	3.64 (4)	4.36 (5)	5.31 (5)	3.81 (4)	4.57 (4)	5.70 (5)
`qLCB`	4.02 (5)	4.24 (4)	4.83 (2)	4.27 (5)	5.04 (5)	5.40 (4)
`rs`	5.57 (6)	5.89 (6)	5.89 (6)	5.17 (6)	5.71 (6)	5.82 (6)

Table 6.1: Ranking for accuracy levels 0.5, 0.1, 0.01 averaged over all benchmarks with `rosenbrock`(·) time function on 4 and 16 CPUs with a time budget of 4 h and 2 h, respectively.

The benchmarks indicate an overall advantage of the new resource-aware MBO algorithm `rambo`: On average, `rambo` is always fastest on 4 and 16 CPUs. `rambo` is closely followed by the asynchronous MBO variant `asyn.ei.bel` for accuracy levels 0.5 and 0.1 on 4 CPUs but the lead becomes more clear on 16 CPUs, especially for the highest accuracy level 0.01. In comparison to the conventional synchronous MBO approaches `ei.bel` and `qLCB`, `rambo` as well as `asyn.eei` and `asyn.ei.bel` reach the given accuracy levels in shorter time on 16 CPUs. This is especially true for objective functions that are highly multimodal and thus hard to model (`ackley`(·), `rastrigin`(·)) by the surrogate as seen in Figure 6.13.

Table 6.1 shows that the less expensive `asyn.ei.bel` approach performs better than the computationally demanding `asyn.eei` on 16 CPUs. On 4 CPUs the synchronous `qLCB` approach is faster than the asynchronous approaches for the highest accuracy level 0.01. This result is influenced by the good performance of `qLCB` on functions with a smooth surface, as can be seen in the upper part of Figure 6.13 in the 5 and 10-dimensional version of the `bohachevsky`(·) benchmark.

When comparing the performance of the approaches for the 2-dimensional versus the 10-dimensional versions of the benchmarks, Figure 6.13 clearly shows that the new `rambo` approach outperforms all other approaches at higher dimensional problems compared to lower dimensions. All presented MBO approaches outperform the base-line random search `rs` on almost all benchmarks and accuracy levels.

For a thorough analysis, Fig. 6.14 exemplarily visualizes the mapping of the parallel configuration evaluations (jobs) for all MBO approaches on 16 CPUs for the $5d$ versions of the benchmarks. Each gray box represents the execution time of a job evaluation on the respective CPU. The gaps represent CPU idle time. For the synchronously executed MBO approaches `rambo`, `qLCB` and `ei.bel` the vertical lines represent the end of an MBO iteration. Red boxes indicate that the CPU is occupied with a point proposal.

The necessity of a resource estimation for jobs with varying runtimes becomes obvious, as the synchronous variants `qLCB` and `ei.bel` can cause long idle times by queuing jobs together with large runtime differences. The scheduling in `rambo`

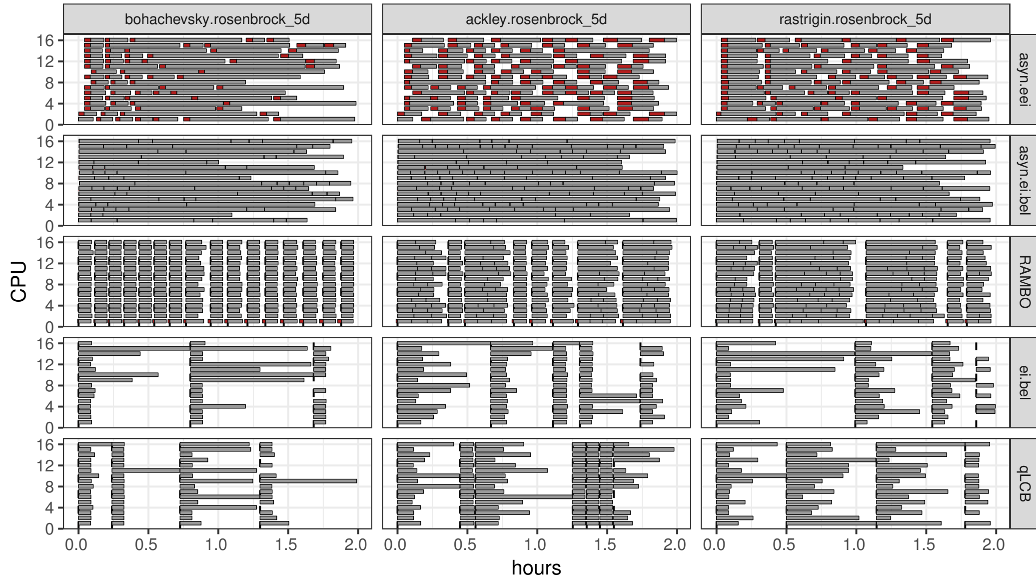

Figure 6.14: Scheduling of MBO algorithms. Time on x-axis and mapping of candidates to $m = 16$ CPUs on y-axis. Each gray box represents a job. Each red box represents overhead of the point proposal for the approaches. The gaps represent CPU idle time.

manages to clearly reduce this idle time. This effect of efficient resource utilization increases with the number of CPUs. `rambo` reaches nearly the same effective resource-utilization as the asynchronous approaches (see Figure 6.14) and at the same time reaches the accuracy level fastest (see Table 6.1).

The Monte Carlo approach `asyn.eei` generates a high computational overhead as indicated by the red boxes, which reduces the effective number of evaluations. Here, the overhead for a new point proposal sometimes needs the same amount of time as the job evaluation. Idling occurs because the calculation of the EEI is encouraged to wait for ongoing EEI calculations to include their proposals. This overhead additionally increases with the number of already evaluated points. In contrast, `asyn.ei.bel` has a comparably low overhead and thus basically no idle time. This seems to be an advantage for `asyn.ei.bel` on 16 CPUs where it performs better on all accuracy levels on average than its computational demanding counterpart `asyn.eei`, especially for higher dimensional problems.

Table 6.2 lists the number of evaluated configurations across all 10 repetitions and for all objective functions, grouped by approach, number of CPUs and dimension. These numbers reflect the scalability of the different MBO approaches. Due to the high overhead of the asynchronous `asyn.eei` approach, its number of evaluated configurations has only a small increase on 16 CPUs versus 4 CPUs. This is especially true for higher dimensional problems (see 4 CPUs 10d column vs 16 CPUs 10d column) where the number of evaluated configurations for the 10-dimensional problems

	4 CPUs			16 CPUs		
Algorithm	2d	5d	10d	2d	5d	10d
asyn.eei	2683	2518	2359	4249	3592	2841
asyn.ei.bel	2976	2676	3974	5485	4291	5978
ei.bel	1649	1974	3255	1815	2241	3927
qLCB	1757	2161	4011	2200	2699	4909
rambo	**4832**	**4260**	**5046**	**8364**	**6535**	**7212**

Table 6.2: Number of evaluated configurations per dimension across all 10 repetitions and objective functions with `rosenbrock(·)` time function on 4 and 16 CPUs with a time budget of 4 h and 2 h, respectively.

is even smaller than the number of evaluations for the conventional synchronous approaches. This indicates that the `asyn.eei` approach has an inefficient resource utilization. In contrast, the `asyn.ei.bel` approach manages to increase the number of evaluations for all dimensions when the number of CPUs increases.

Both synchronous MBO approaches `qLCB` and `ei.bel` have a low number of evaluations compared to the asynchronous `asyn.ei.bel` approach and `rambo` and are also not able to decently increase their number of evaluations when the number of CPUs increases. Overall, the resource-aware `rambo` approach has the highest number of evaluated jobs in all problem dimensions and is also able to scale its evaluations with the number of available CPUs.

If the resource estimation that is used in `rambo` has a high quality, `rambo` clearly outperforms the conventional synchronous MBO approaches. This indicates that the resource utilization obtained by the scheduling in `rambo` leads to faster and better results, especially, when the number of available CPUs and the dimension of a problem (number of configurable parameters) increases. On average `rambo` converges faster to the optimum than all considered asynchronous approaches.

Evaluation with Low Resource Estimation Quality

The time function `rastrigin` used in the following scenario is difficult to fit by the runtime regression model, as visualized by the left residual plot in Figure 6.12. For this reason, the benefit of our resource-aware strategy is expected to be minimal. For example, in a possible worst case multiple supposedly short jobs are assigned to one CPU but their real runtime is considerably longer leading to a delayed model update.

Similar to the previous subsection, Figure 6.15 shows box plots for the benchmark results, but with `rastrigin(·)` as the time function. Table 6.3 provides the mean ranks for Figure 6.15 and Table 6.4 the number of evaluated configurations, calculated in the same way as in the previous evaluation with high resource estimation quality.

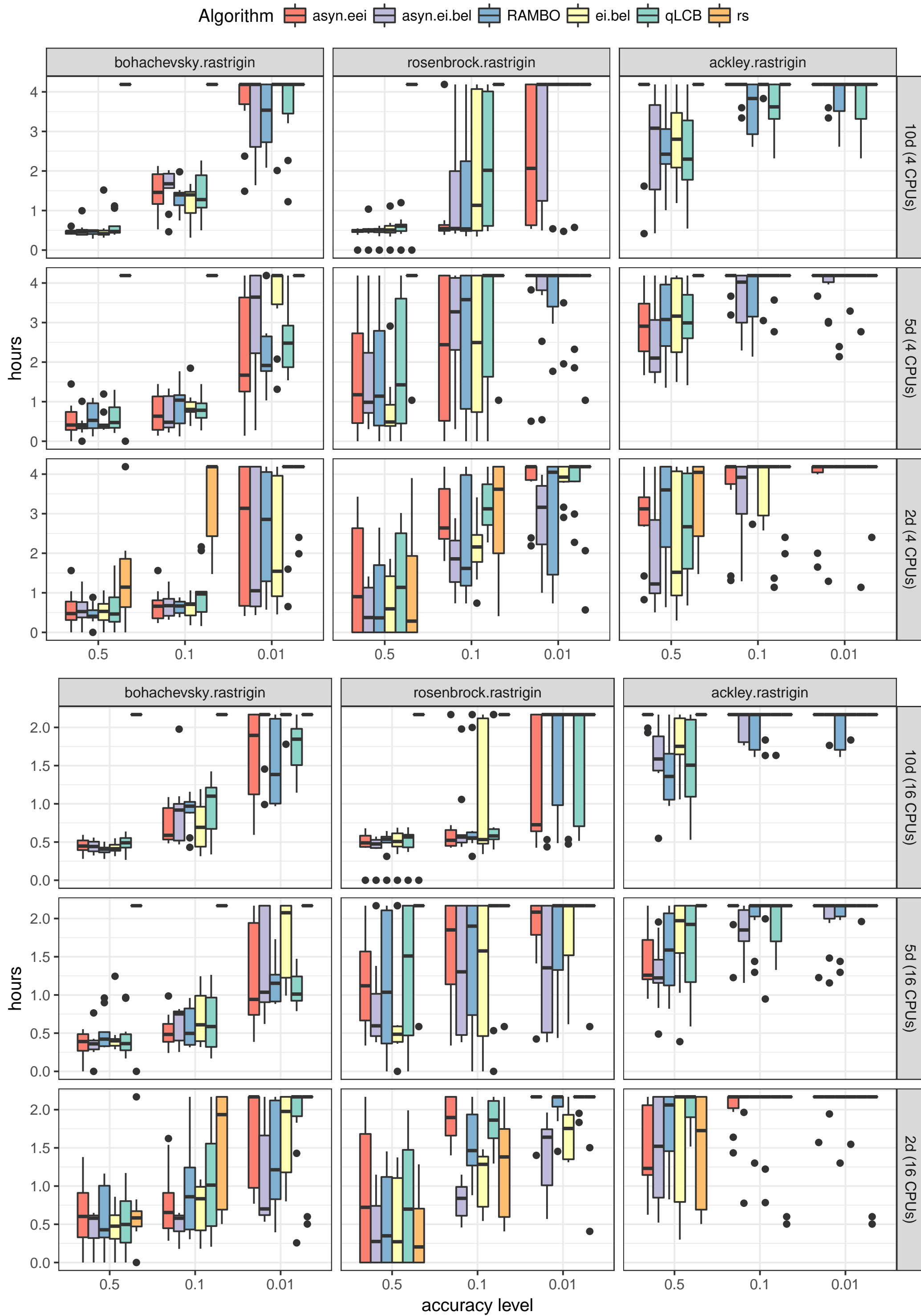

Figure 6.15: Accuracy level vs. execution time for different objective functions using time function `rastrigin`(·) (lower is better).

Despite possible wrong scheduling decisions, `rambo` still manages to outperform the synchronous `qLCB` approach and also performs better than `ei.bel` and `asyn.eei` on the highest accuracy level 0.01 on average (see Table 6.3).

	4 CPUs			16 CPUs		
Algorithm	0.5	0.1	0.01	0.5	0.1	0.01
`asyn.eei`	3.78 (4)	3.93 (4)	4.52 (3)	3.91 (4)	4.02 (3)	4.58 (3)
`asyn.ei.bel`	2.98 (1)	3.48 (1)	4.30 (1)	3.06 (1)	3.19 (1)	4.13 (1)
`rambo`	**3.50 (3)**	**3.69 (2)**	**4.40 (2)**	**3.47 (2)**	**4.04 (4)**	**4.17 (2)**
`ei.bel`	3.19 (2)	3.83 (3)	5.08 (5)	3.61 (3)	3.67 (2)	4.79 (4)
`qLCB`	3.79 (5)	4.13 (5)	4.83 (4)	4.01 (5)	4.49 (5)	4.88 (5)
`rs`	5.56 (6)	5.61 (6)	5.70 (6)	5.26 (6)	5.41 (6)	5.64 (6)

Table 6.3: Ranking for accuracy levels 0.5, 0.1, 0.01 averaged over all problems with `rastrigin`(·) time function on 4 and 16 CPUs with a time budget of 4 h and 2 h, respectively.

This is also reflected in Table 6.4: `rambo` has executed more jobs compared to the `qLCB` and `ei.bel` approach in most cases. For the 10-dimensional problems `rambo` also evaluated more configurations than the `asyn.eei` approach and closely follows the number of evaluations of the `asyn.ei.bel` approach. In comparison to Table 6.2, where the `rosenbrock`(·) time function was used, here the `rastrigin`(·) time function produces less job evaluations on all approaches in general. Similar to the previous benchmarks with the `rosenbrock`(·) time function, the simplified `asyn.ei.bel` seems to benefit from its reduced overhead and places first on 4 and 16 CPUs, where it is closely followed by `rambo` for the highest accuracy level 0.01 (see Table 6.3).

	4 CPUs			16 CPUs		
Algorithm	2d	5d	10d	2d	5d	10d
`asyn.eei`	1348	1512	1919	2050	2195	2338
`asyn.ei.bel`	1304	1588	2116	2178	2316	2831
`ei.bel`	1041	1391	1987	1537	1833	2512
`qLCB`	1034	1462	1978	1593	1918	2461
`rambo`	**1128**	**1532**	**2023**	**1589**	**2127**	**2549**

Table 6.4: Number of evaluated configurations per dimension across all 10 repetitions and objective functions with `rastrigin`(·) time function on 4 and 16 CPUs with a time budget of 4 h and 2 h, respectively.

Overall, `rambo` appears not to be able to outperform the asynchronous `asyn.ei.bel` approach as unreliable runtime estimates likely lead to suboptimal scheduling decisions. However, `rambo` reaches comparable results to `asyn.eei` and compared to the conventional synchronous approaches it is a viable choice.

Summary

The comparison study of the new resource-aware scheduling for MBO included in the RAMBO framework against the popular synchronous and asynchronous MBO approaches on a set of illustrative test functions for global optimization methods shows that `rambo` was able to outperform the synchronous MBO approach `qLCB` on the benchmark functions. On setups with high runtime estimation quality `rambo` converged faster to the optima than the competing MBO approaches on average, which is especially true for higher dimensional problems (higher number of possible parameter configurations). This indicates that the resource utilization obtained by the approach developed in this thesis improves MBO, especially, when the number of available CPUs increases.

On setups with low runtime estimation quality, the simplified asynchronous Kriging believer performed best. Unreliable estimates likely lead to suboptimal scheduling decisions for `rambo`. While the asynchronous Kriging believer approach and `rambo` benefited from increasing the number of CPUs, the overhead of the asynchronous approach based on EEI increased.

If the runtime of point evaluations is predictable the new `rambo` approach is suggested for parallel MBO with high numbers of available CPUs and higher dimensional problems. Even if the runtime estimation quality is obviously hard to determine in advance, for real applications like hyperparameter optimization for machine learning methods, predictable runtimes can be assumed. The results also suggest that on some setups the choice of the infill criterion determines which parallelization strategy can reach a better performance.

6.4 Resource-Aware Scheduling Strategies for Heterogeneous Embedded Architectures

The above-described scheduling strategies are not only applicable for homogeneous computer architectures, but also for heterogeneous ones. This section presents the approach of resource-efficient execution of parallel MBO on heterogeneous architectures via RAMBO. Heterogeneous architectures are commonly found in mobile embedded devices. Such devices typically consist of different processors with different frequencies and memory sizes and are characterized by tight resource and energy restrictions. When MBO is executed on heterogeneous systems, the execution time of the evaluation of configurations can vary heavily not only depending on the configuration itself – as was shown for homogeneous architectures – but also depending on which processor an evaluation is executed.

The original `parallel` package [R C17a] that is part of the R language targets problems that can be decomposed into independent tasks that are then processed in parallel. While sufficient for parallelizing MBO on homogeneous architectures, it lacks dedicated support for heterogeneous architectures. To address this shortcoming, the parallel package was enhanced by Kotthaus et al. [KLN+17] to support heterogeneous

architectures. This change has already been upstreamed into the R language[4] [R C17a].

The key to the heterogeneous MBO approach is a resource utility estimation that uses a regression model that estimates the execution times of a configuration for each available processor type. In combination with an extended knapsack-based scheduling strategy and the enhancement of the `parallel` package that allows for allocating tasks to specific processors, this yields a resource-aware scheduling of MBO tailored for heterogeneous architectures.

Subsection 6.4.1 presents the extension of the knapsack based scheduling strategy (described in Subsection 6.3.2) including the new resource utility estimation model. The evaluation of this approach is described in Subsection 6.4.2.

6.4.1 Knapsack based Scheduling Strategy for Heterogeneous Architectures

As described in Section 6.3 the resource-aware scheduling for MBO uses two inputs: the estimated resource utilization and the priority of the proposed candidates. While the priority of a candidate is computed as described in Section 6.3, the estimation of the resource utilization needs to be enhanced for heterogeneous systems.

Resource Utility Estimation for Heterogeneous Systems

The regression model that is used to estimate the execution times of the candidates was previously based on Kriging, now Random Forest is applied instead. Random Forest is more suitable for heterogeneous systems since the job execution times build up a discontinuous model due to the addition categorical variable that represent the processor type.

The regression model now needs to estimate the runtime $\hat{t}_j$ for each candidate in the proposed set of jobs $J = \{1, \ldots, q\}$ per available CPU $K = \{1, \ldots, m\}$, since the execution of a job might result in different execution times depending on which processor type the job is scheduled on. If the underlying heterogeneous architecture is known, the number of runtime estimates per job can be reduced to the number of different processor types. Thus the runtime of a job $j \in J$ is predicted for each available processor type $k \in K$ in each MBO iteration based on the runtime of all previously evaluated jobs to build the runtime model of the black-box function and is therefore denoted as $\hat{t}_{kj}$.

Knapsack based Scheduling Strategy for Heterogeneous System

To apply the $0 - 1$ multiple knapsack algorithm for the scheduling on heterogeneous architectures the original formulation from Section 6.3.2 needs to be extended.

[4] Kotthaus et al.: Support for Heterogeneous Processors - Section 8. Setting the CPU Affinity with mclapply: `https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf`,2018

Now the items that represent the jobs J, have different weights represented by the different runtime estimates $\hat{t}_{jk}$ per processor type k. Since the capacity of the CPUs is now heterogeneous, a reformulation is needed. For this purpose, a ratio variable that represents an approximated ratio of the runtime differences produced by the different processor types is introduced.

To minimize the delay of the model update with the results of the most promising candidate, the job with the highest priority $j^* := \arg\max_j p_j$ is now always placed on the CPU $k^* := \text{argmin}_k \hat{t}_{kj*}$ that leads to the shortest estimated runtime for j^*. The capacity for the remaining CPUs and thus the time bound for each MBO iteration is accordingly defined by the shortest estimated runtime of the highest prioritized job $\hat{t}_{k^*j^*}$. Additionally, the ratio variable $\hat{t}_{k^*j^*}/\hat{t}_{kj^*}$ that represents the runtime difference of the highest prioritized job on the remaining k CPUs is introduced.

Note that in general, the assumption that runtimes on different CPU types differ by a constant factor refers to the uniform processor model described by Pinedo, which is a simplified model of real hardware [Mar18]: for example, one CPU might offer vector instructions that some jobs heavily benefit from, whereas others barely make use of them. Instead of relying on statically precomputed ratios (e.g., derived from the ratio of CPU frequencies), the selected job j^* is used as the "benchmark" for comparing CPU speeds in a given MBO iteration, under the assumption that in this iteration the speed on CPU k differs from k^* by a factor of $\hat{t}_{k^*j^*}/\hat{t}_{kj^*}$. The formulation of the restriction of the capacities for the remaining CPUs is thus as follows, while the rest of the knapsack algorithm remains as described in 6.3.2:

$$\hat{t}_{k^*j^*} \frac{\hat{t}_{k^*j^*}}{\hat{t}_{kj^*}} \geq \sum_{j \in J} \hat{t}_{k^*j} c_{kj} \;\forall k \in K. \tag{6.8}$$

Here, the estimated execution times of the remaining candidates on the fastest CPU $\hat{t}_{k^*j}$ on the right hand side of equation (6.8) is expected to be approximately similar to the estimated runtime of a job on the remaining CPUs $\hat{t}_{kj}$ multiplied with the ratio variable:

$$\hat{t}_{k^*j} \approx \hat{t}_{kj} \frac{\hat{t}_{k^*j^*}}{\hat{t}_{kj^*}} \;\forall k \in K, \forall j \in J. \tag{6.9}$$

This formulation is needed to reduce the number of weights (number of runtime estimates per CPU type) per item j to a single weight $\hat{t}_{k^*j}$ in order to apply the original knapsack algorithm.

In the next section, the RAMBO framework including the above-described scheduling strategy will be evaluated.

6.4.2 Evaluation

The effectiveness of the heterogeneous RAMBO approach is evaluated by targeting the ARM big.LITTLE architecture[5] of the Odroid-XU3 platform[6] that is also commonly found in mobile devices [KLN+17]. This platform is equipped with four "big" Cortex A15 CPUs (quad-core) with a frequency that can be scaled up to 2.0 GHz and four "little" Cortex A7 CPUs that have about half the processor speed (1.4 GHz). The Odroid-XU3 platform also includes a Mali-T628 GPU (not considered for the evaluation) and 2 GB of main memory.

For the evaluation of RAMBO on heterogeneous processing architectures, not only the runtime that is needed to find the best possible configuration is examined but also the energy consumption. This is accomplished by reading from the power measurement sensors INA231 offered by the Odroid-XU3 platform, which report energy consumption for both processor types as well as for the RAM and the GPU. To measure the energy consumption of the resource-aware scheduling strategy and its competing MBO approaches, a so-called Relay Reader [Neu17] is used to read out the sensor data in regular intervals of approximately one second via threads for both CPU types. These threads are executed on separate CPUs and do not influence the runtime measurements of the MBO approaches.

A subset of the setup described in Subsection 6.3.4 is used as the experimental setup. RAMBO will be compared to the conventional synchronous MBO approach that uses the qLCB multi-point infill criterion (6.1) and to the asynchronous MBO approach that aims at using all available CPU time to solve the optimization problem in parallel and uses the Kriging believer criterion [GLC10]. All MBO approaches are evaluated on the 2-dimensional versions of the synthetic functions presented in Subsection 6.3.4 and executed on four CPUs, two of each type (i.e., Cortex A15 and Cortex A7).

The runtime of the objective functions was previously simulated by sleeping for a given time, determined via an additional synthetic function, that represented the runtime behavior of the respective objective function. For the energy measurements, a real computation is needed. This is accomplished by repeatedly executing a function that draws random numbers. The runtime of this real computation is still controlled via an additional synthetic function that defines the number of repetitions and thus simulates the time that is needed for calculating the objective value. For the synthetic function that simulates the runtime of the objective functions the `rosenbrock`(d) function is used, since it delivers a more reliable runtime estimation than `rastrigin`(d) (see Subsection 6.3.4). The output of the `rosenbrock`(2) function is scaled to return values between 5 min to 50 min.

---

[5] ARM big.LITTLE Technology: `https://developer.arm.com/technologies/big-little`, 2018

[6] Odroid-XU3: `https://developer.arm.com/graphics/development-platforms/odroid-xu3`, 2018

The MBO approaches run for 2 h on $m = 4$ CPUs including all computation overhead and CPU idling. The initial set is generated as for the homogeneous experiments by the latin hypercube sampling [MBC00] with $n = 4 * d$ configurations and all of the following approaches start with the same initial set in all 10 repetitions:

- **`asyn.ei.bel`:** Asynchronously executed MBO approach using Kriging believer.
- **`qLCB`:** Synchronously executed MBO approach using qLCB where in each MBO iteration $q = m$ configurations are proposed.
- **`rambo`:** New synchronously executed MBO approach using qLCB with the knapsack based resource-aware scheduling strategy for heterogeneous systems, where in each iteration $q = 3 \cdot m$ candidates are proposed.

Evaluation of the MBO Performance

Table 6.5 lists the aggregated ranks over all 2-dimensional objective functions, grouped by accuracy level. As described in Section 6.3.4, the approaches are ranked w.r.t. their performance for each of the 10 repetitions as well as each benchmark before they are aggregated into the mean.

Figure 6.16 shows the corresponding box plots for the time required to reach the three different accuracy levels, as described in Subsection 6.3.4. The faster an approach reaches the desired accuracy level, the lower the displayed box and the better the approach.

Algorithm	0.5	0.1	0.01
`rambo`	**1.90 (1)**	**1.77 (1)**	**1.90 (1)**
`asyn.ei.bel`	2.07 (2)	2.43 (2)	2.63 (2)
`qLCB`	2.67 (3)	2.63 (3)	2.70 (3)

Table 6.5: Ranking for accuracy levels 0.5, 0.1, 0.01 averaged over all problems with `rosenbrock`(2) time function on 4 CPUs with a time budget of 2 h.

The benchmarks indicate an overall advantage of the new knapsack based algorithm for heterogeneous systems, especially for the highest accuracy level 0.01 (see Figure 6.16). On average, `rambo` is always fastest in reaching each of the three accuracy levels and thus converges faster to the optimum in the given time budget of 2 h (see Table 6.5).

Figure 6.16 shows that in comparison to `rambo`, the conventional synchronous MBO approach `qLCB` is not able to reach the highest accuracy level 0.01 for the `rastrigin`(2) and `ackley`(2) functions within all of the 10 repetitions. The same can be said about the asynchronous MBO approach `asy.ei.bel` for the `bohachevsky`(2) and `rastrigin`(2) functions.

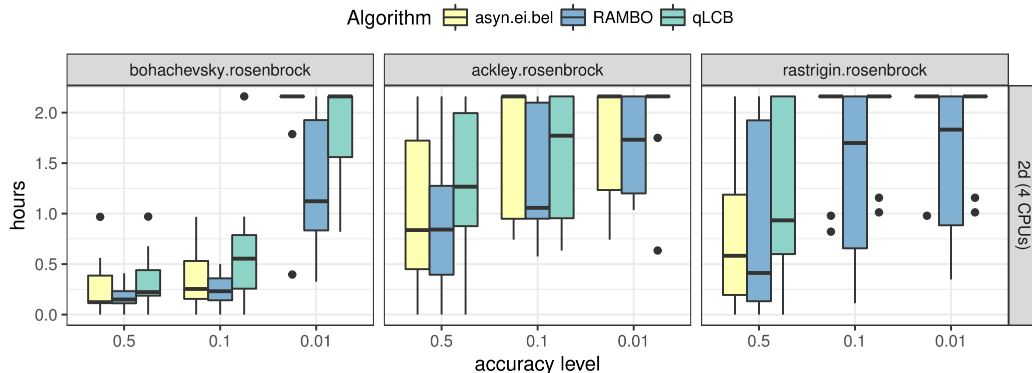

Figure 6.16: Accuracy level vs. execution time for the 2-dimensional objective functions using time function `rosenbrock`(2) (lower is better).

Table 6.6 lists the number of evaluated configurations and the number of MBO iterations in brackets over all 10 repetitions for each objective function. The asynchronous approach `asy.ei.bel` has no MBO iterations, it desynchronizes the model update (see Subsection 6.1.2) aiming at using all available CPU time to solve the optimization problem.

Algorithm	`bohachevsky`	`ackley`	`rastrigin`
`rambo`	**407 (131)**	**431 (148)**	**403 (108)**
`asyn.ei.bel`	415	399	380
`qLCB`	331 (66)	340 (65)	297 (56)

Table 6.6: Number of evaluated configurations and performed model updates in brackets (for the synchronous MBO approaches), across all 10 repetitions for each objective functions with `rosenbrock`(2) time function on 4 CPUs with a time budget of 2 h.

The knapsack based scheduling for heterogeneous systems included in `rambo` always manages to evaluate a higher amount of configurations compared to the `qLCB` approach within the given time budget. When looking at the number of performed MBO iterations, `rambo` outperforms `qLCB` by a factor of almost two.

The `asy.ei.bel` approach aims to produce no CPU idle time by desynchronizing the model update. For the `rambo` approach, the CPU idle time can occur in cases where the selected configurations do not ideally fit into the MBO iteration time bound, which is defined by the runtime of the configuration with the highest priority. However, within the given time budget, the number of evaluations performed by `rambo` is at least very similar to the number of evaluations performed by `asy.ei.bel`, and on most benchmarks even exceeds it.

Since `rambo` evaluates more configurations compared to `qLCB` and `asy.ei.bel` for most of the benchmarks within the same time budget, it might also consume

more energy. The energy consumption of all three approaches is therefore analyzed in the next section.

Evaluation of the Energy Consumption

Figure 6.17 shows the box plots for the energy consumption over all 10 repetitions for each benchmark on each CPU type (upper part, Cortex `A7` and Cortex `A15`) and over all CPUs (lower part, `combined`). The less energy an approach consumes, the lower the displayed box.

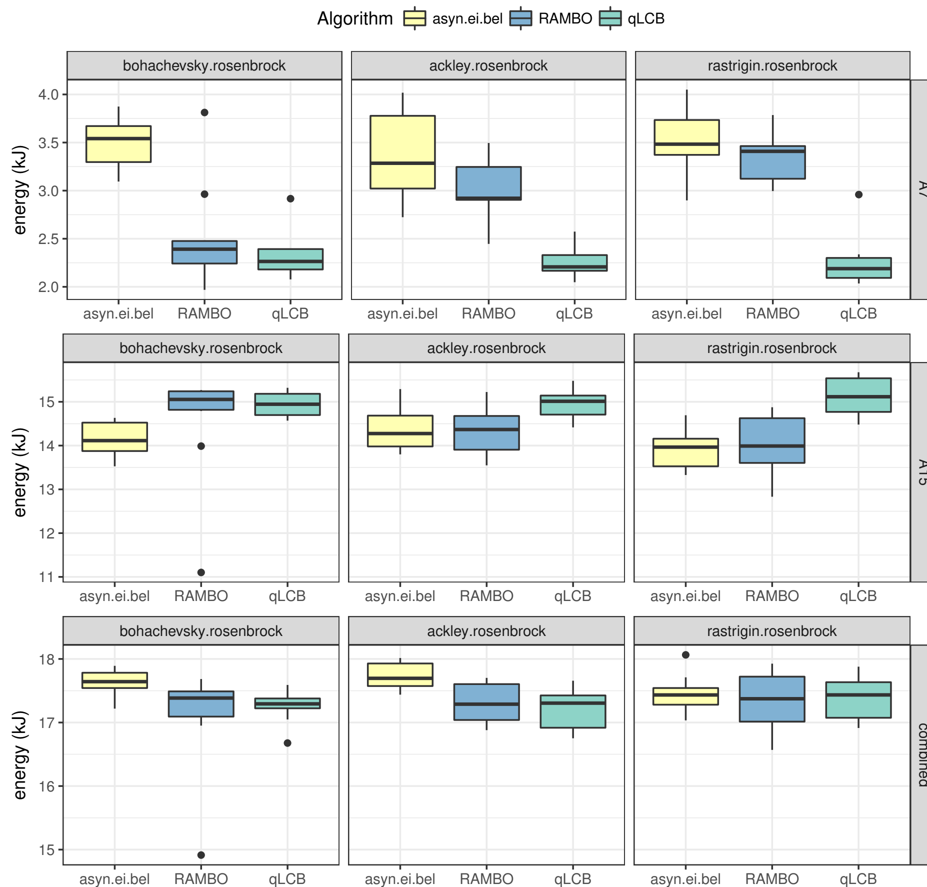

Figure 6.17: Energy consumption in kJ on the two A15 CPUs (2.0 GHz), the two A7 CPUs (1.4 GHz) and combined consumption on both CPU types across all 10 repetitions for each objective function, with `rosenbrock`(2) time function and a time budget of 2 h (lower is better).

The results in Figure 6.17 indicate that `rambo` consumes more energy compared to the default `qLCB` approach on the "slow" Cortex `A7` CPUs, while it consumes less energy on the "fast" Cortex `A15` CPUs. In comparison to the `asy.ei.bel` approach, `rambo` manages to consume less energy on the "slow" Cortex `A7` CPUs.

The reason for the higher energy consumption of `rambo` compared to the synchronous `qLCB` approach on the "slow" Cortex `A7` CPUs (see upper part of Figure 6.17) lies in the resource-aware scheduling strategy that is able to utilize the less energy consuming `A7` CPUs more efficiently by mapping jobs to specific CPUs. Furthermore, only jobs with a runtime smaller or equal to the job with the highest priority are executed within one MBO iteration, thus longer running jobs with a lower optimization potential are discarded and more MBO iterations can be performed in the given time budget. In contrast, `qLCB` is not able to map jobs to specific CPUs, it just starts four jobs on the 4 available CPUs that were proposed by the infill criterion in each MBO iteration, without respect to the heterogeneity of the underlying architecture and the job execution times.

Another contributing factor to the higher energy consumption of the `qLCB` approach is that it executes more jobs on the more energy-consuming `A15` CPUs due to the OS scheduling. Within one MBO iteration, the OS scheduler migrates jobs from a "slow" `A7` CPU to a "fast" `A15` CPU, for cases where a job on a fast CPU finishes earlier than a job on a slow CPU, to speed up computation and thus execute more MBO-iterations. Hence, `qLCB` has nearly no idle time on the `A15` CPUs. However, the conventional synchronous approach does only perform approximately half as many MBO iterations as `rambo` (see Table 6.6).

As shown in Table 6.6, `rambo` in general executed more job evaluations in the given time compared to both competing MBO approaches. However, the `combined` energy consumption on all four CPUs depicted in the lower part of Figure 6.17 shows that `rambo` consumes approximately the same amount of energy as `qLCB`, while it consumes less energy compared to `asy.ei.bel` for the `bohachevsky`(2) and `ackley`(2) benchmark functions.

The asynchronous `asy.ei.bel` approach in most cases consumes more energy than `rambo` since it has nearly no CPU idle time, however, it still converges slower to the optimum. The reason for this is that `rambo` is able to select more promising candidates with shorter runtimes since it only executes jobs with a shorter or equal runtime than the most promising candidate and thus aims at finding the cheapest way of evaluations through the model.

Overall, the results show that the resource utilization obtained by the scheduling for heterogeneous architectures in `rambo` enables MBO to converge faster to the optimum, without consuming more energy resources than the competing approaches.

6.5 Summary

This chapter presented new resource-aware scheduling strategies for parallel machine learning algorithms. While there are many different parallel algorithms, this chapter focused on the optimization of parallel MBO, a state-of-the-art global optimization method for expensive black-box functions with huge resource demands. The goal was to execute MBO in a resource-efficient way to enable the processing of larger

problem sizes within a given time budget or, in other words, reduce the end-to-end wall-clock time for a constant problem size.

In contrast to classical scheduling problems, the scheduling for MBO needs to interact with the configuration proposal mechanism to select black-box configurations with suitable resource demands for parallel evaluation, which is a complex problem since the resource demands need to be known (at least estimated) before execution.

For this purpose, a new resource-aware model-based optimization framework called RAMBO was presented and evaluated. With RAMBO it becomes possible to make use of the full potential of parallel architectures in an efficient manner. This was accomplished by developing an estimation model for the runtimes of each evaluation of a black-box function to guide the scheduling of configurations to available resources. In addition, an execution priority reflecting the estimated profit of a black-box evaluation was used to guide MBO to interesting regions in a faster and resource-efficient way without directly favoring less expensive configurations.

The presented scheduling strategies aim at guiding MBO to interesting regions in a faster and resource-aware way, to acquire the feedback of the parallel worker processes that evaluate the black-box configurations, in the shortest possible time to avoid model update delay while reducing the CPU idle time.

Two different scheduling strategies were proposed and analyzed, the First Fit scheduling strategy and the knapsack based scheduling strategy. The results for the First Fit scheduling strategy showed that RAMBO managed to balance long execution times more evenly and thus executes more evaluations in the same time budget, leading to a higher confidence in the optimization space compared to conventional synchronous parallel execution model. The knapsack based scheduling approach was compared to existing parallel synchronous MBO approaches and to approaches that aim at reducing the idle time by asynchronously updating the model. Here, the results showed that RAMBO converged faster to the optimum than the existing approaches in cases where the resource estimates were reliable. The concept of RAMBO was especially efficient for complex high dimensional problems and also strongly improved upon the existing approaches in terms of scalability when the number of available CPUs was increased.

Furthermore, the knapsack based scheduling strategy was enhanced to optimize parallel MBO on heterogeneous architectures that are commonly found in embedded systems. The results showed that RAMBO converged faster to the optimum compared to the synchronous and asynchronous parallel MBO approaches. Thanks to a more efficient resource utilization obtained by the scheduling, RAMBO managed to evaluate more configurations in the given time budget without consuming more energy than the competing approaches.

Chapter 7

Conclusion and Outlook

In the preceding chapters of this thesis, it has been shown that running machine learning algorithms like model-based optimization in homogeneous or heterogeneous, resource-constrained environments poses a number of challenges that classical approaches fail to address adequately. RAMBO has been proposed as a potential means to a more efficient utilization of resources in such environments. RAMBO significantly outperforms other approaches, meaning that it can find better algorithm configurations with better prediction quality in a shorter amount of time. RAMBO is a significant step forward to reaching the goal of efficient resource utilization for statistical machine learning algorithms, especially for the execution on small and computationally weak devices that surround us and make our lives better or - in the case of medical devices - might even work to save them.

7.1 Summary of Research Contributions

This thesis started off by introducing the concept of model-based optimization (MBO), an important global optimization approach with huge resource demands, which can be applied for automatically configuring parameterized algorithms, as well as giving an overview of the R programming language, the de facto standard software environment for the development of statistical learning applications.

The potential for new optimizations that enable statistical learning algorithms to scale to larger problem sizes was analyzed in Chapter 3. Previously published optimization approaches have usually shown improvements for simple R programs or specific R functions, but they exhibited fairly mixed results when it came to speeding up complex real-world applications like machine learning algorithms. Building upon the observation that real-world programs are different from synthetic benchmarks, the most common classification algorithms were analyzed combined with real-world input data sets to identify the main reasons for their lavish use of resources. On the basis of the redesigned and enhanced traceR profiling framework, detailed insights into the runtime and memory behavior of these algorithms were provided.

The analysis with traceR focused on learning algorithms, both alternative R implementations as well as the original GNU R language can use the results to guide the development of optimizations that improve the resource utilization of real-world code. Overall, the conducted analysis demonstrated that memory management is one of the major contributors to the total runtime of the algorithms. It was shown that the time spent in memory management is inflated by wasteful memory

allocation policies of the R execution environment, as well as by the footprint and the number of allocated vector data structures. Here, especially vector data structures that span multiple pages of memory dominated. The results suggested a general memory optimization to reduce the memory footprint and thus the runtime needed for memory allocation and garbage collection.

In Chapter 4, the results of the analysis from Chapter 3 were used to develop an optimization for efficient memory utilization. This optimization is application-transparent and employs page sharing at a memory management layer between the R interpreter and the operating system's memory management. The optimization benefits a large number of applications since it preserves compatibility with the available software libraries that most statistical programs are based on, and covers one of the most important resource bottlenecks of machine learning algorithms. It avoids duplication of page contents for large vector data structures and optimizes the copy-on-write mechanism of the R language. By concentrating on the most rewarding optimizations - the sharing of zero-filled pages and deduplicating at the page level instead of the object level -, the overhead of more general OS level memory optimization approaches such as deduplication and compression is avoided. The proposed optimization achieved a considerable reduction of the memory consumption by up to 53.5% with an average of 18% for a large number of typical real-world benchmarks. It also significantly speeds up the computation up to a factor of 5.2 in cases where previously pages had to be swapped out due to insufficient main memory.

In addition to the proposed memory optimization, the second major avenue for optimizing statistical machine learning algorithms that was explored in this thesis is parallelization, which poses new resource utilization challenges. In order to fully benefit from parallel execution, the bottlenecks arising in embarrassingly parallel applications were analyzed in Chapter 5. This was enabled by enhancing the R profiling framework traceR for the analysis of parallel programs. For hyperparameter optimization, the results showed that a high runtime variance in the configuration space causes inefficient resource utilization due to different completion times of evaluations running in parallel. Overall, the results showed that the parallel computation methods provided by R packages are not sufficient and can lead to inefficient resource utilization, which calls for the development of new resource efficient parallelization strategies, including new scheduling strategies.

Chapter 6 of this thesis focused on the development of resource-aware scheduling strategies for parallel machine learning applications. While there are many different parallel algorithms, this thesis focused on the optimization of parallel MBO, a state-of-the-art global optimization method for expensive black-box functions with huge resource demands. To efficiently map the evaluations of black-box configurations to the underlying parallel architecture, depending on their resource demands, new scheduling strategies are needed. In contrast to classical scheduling problems, the scheduling for MBO needs to interact with the configuration proposal mechanism to select black-box configurations with suitable resource demands for efficient parallel

evaluation, which is a complex problem since the resource demands need to be known (at least estimated) before execution. For this purpose, a new resource-aware model-based framework called RAMBO was presented.

With RAMBO and its integrated scheduling strategies, it becomes possible to make use of the full potential of parallel architectures in an efficient manner. The scheduling strategies presented in this thesis are aiming at guiding MBO to interesting regions in a faster and resource-aware way. The goal was to acquire the feedback of the parallel worker processes that evaluate the black-box configurations in the shortest possible time to avoid MBO model update delay while reducing the idle time on the workers. Therefore, a model that estimates the runtime for each evaluation of a black-box function has been developed to guide the scheduling of configuration evaluations to available resources. In addition, an execution priority reflecting the estimated profit of a black-box evaluation was used to guide MBO to interesting regions in a faster way without directly favoring less expensive configurations.

Two scheduling strategies have been proposed and extensively evaluated, the First Fit scheduling strategy and the knapsack based scheduling strategy. The results for the First Fit scheduling strategy showed that RAMBO managed to balance long execution times more evenly and thus executed more evaluations in the same time budget, leading to a higher confidence in the optimization space compared to the conventional synchronous MBO execution model.

The knapsack based scheduling approach has been compared to existing parallel synchronous MBO approaches and to approaches that aim at reducing the idle time by asynchronously updating the model. Here, the results showed that RAMBO converged faster to the optimum than the existing approaches in cases where the resource estimates were reliable. The concept of RAMBO was especially efficient for complex high-dimensional problems and also strongly improved upon the existing approaches in scalability performance, when the number of available CPUs was increased.

Furthermore, the knapsack based scheduling strategy has been enhanced to optimize parallel MBO on heterogeneous architectures that are commonly found in embedded systems. The results showed that RAMBO is also able converged faster to the optimum on heterogeneous architectures compared to the exiting approaches. Thanks to the resource utilization obtained by the scheduling, more configurations were evaluated in the given time budget while consuming the same amount of energy compared to the competing approaches.

Overall, RAMBO significantly outperforms other parallel MBO approaches, meaning that it finds better algorithm configurations in a shorter amount of time. For reaching the goal of efficient resource utilization in statistical machine learning algorithms, RAMBO is thus a significant step forward.

7.2 Future Research

Extended Profiling: The data produced by the traceR profiling tool can be utilized to enable profile-driven optimizations and thus avoid overhead of unnecessary optimizations. For example, the R runtime environment offers the possibility of dynamic compilation that can be enabled and disabled during runtime and thus could be guided by profiling data. The traceR framework itself could be extended for a precise analysis of read and write behavior on large data structures for applying the memory optimization presented in Chapter 4 more selectively and thus reducing its runtime overhead.

Memory Optimization Opportunities: The memory optimization presented in Chapter 4 includes a static refinement (see Section 4.3.2) that disables the optimization for objects above a size limit of two pages, which avoids overhead for small objects where the optimization potential is low. To further reduce this overhead, machine-learning techniques could be applied to optimize the trade-off between runtime and memory, since the parameters of the approach allow for dynamic tuning.

Resource-Aware MBO Optimization: In Chapter 6 a comprehensive framework for resource-aware model-based optimization has been presented that can be used for future work, guiding the development of new resource-aware optimizations. Further work for RAMBO could concentrate on integrating a memory estimation model. As shown in Section 5.3.3 the memory usage heavily influences runtime if the amount of available main memory in the system is too small to hold all required data. Here the traceR profiling tool presented in Sections 3.2 and 5.2 could be applied to gather profiling information of memory usage during runtime that can then be leveraged to guide the scheduling decisions for experiments with high memory demands.

Parallel MBO can be executed synchronously or asynchronously as described in Sections 6.1.2 and 6.1.1. RAMBO already includes resource-aware scheduling strategies for synchronous parallel execution. As shown in the evaluation results of Section 6.3.4, the asynchronous execution has several disadvantages, such as high computational overhead caused by frequent model updates performed after each evaluation. For future work, a resource-aware strategy for the asynchronous approach might be helpful to reduce its overhead. This could be accomplished by integrating the runtime estimation model into the asynchronous execution model, to control the frequency of model updates. By utilizing the runtime estimates of busy job evaluations, a model update could be delayed in cases where the time needed to finish a busy job and model updates performed on other CPUs is very short. This would reduce the number of unnecessary model updates and also improve the candidate selection since the model is enriched with more results.

Resource-Aware Scheduling Strategies for MBO: In comparison to existing synchronous MBO approaches (see Section 6.1.1) the resource-aware scheduling strategies developed in this thesis, reduce the CPU-idle time for an efficient utilization of parallel architectures. However, CPU idling can still occur. The MBO iteration time bound is determined by the estimated runtime of the job with the highest priority to avoid model update delay with the most promising result. The remaining candidates that are proposed to run in parallel with the best candidate are discarded if they do not meet this time bound (see Section 6.3.2). Candidates with a smaller runtime than this time bound can thus produce idle time. To avoid CPU idling, a new scheduling strategy that is able to migrate jobs during runtime to other CPUs might be useful for future work. For this propose, the R programming language has to be extended with a job migration mechanism to allow jobs to be executed on different CPUs during the optimization process. If jobs can be migrated during execution the CPU idle time can be further reduced by enhancing the knapsack based scheduling.

Multi-Objective Candidate Selection for MBO: The performance and resource utilization of MBO is influenced by the selection of the candidates, proposed for evaluation by the infill criterion. The infill-criterion has the goal to reduce the number of necessary black-box evaluations by proposing the most promising candidates to find the best configuration in the shortest possible time. Furthermore, the resource-aware scheduling strategies presented in Chapter 6 are applied to optimize the parallel execution of the proposed candidates, to guide MBO in a resource-efficient way to the optimum in a given time bound. To optimize the selection of the candidates with respect to their resource demands and their performance in one step, a multi-objective infill criterion might be helpful for future work. The infill criterion quantifies the improvement of a candidate evaluation based on a compromise between good predicted outputs and uncertainty about the search space region (a high potential to optimize the quality of the regression model).

For future work, a group-infill criterion could be developed that proposes different sets of candidates and thus assigns one priority to each set. This criterion can be based on the available hardware resources, the runtime variance of the candidates, that lead to different amounts of CPU-idle time, the memory consumption of these candidates, the distance of the candidates in the model space to avoid evaluations of similar configurations and the predicted outputs as well as the uncertainty about the search space, in order to find the cheapest way through the model. With such an infill criterion the resource-awareness would be directly integrated with the configuration proposal mechanism. For this purpose, an extensive analysis would be required to figure out the amount of influence each objective should have in order to build a group priority.

Bibliography

[ADK06] K. Alexandros, M. David, and H. Kurt. "Support Vector Machines in R". In: *Journal of Statistical Software* 15.1 (2006), pp. 1–28 (Cited on pages 79, 97).

[AEW09] A. Arcangeli, I. Eidus, and C. Wright. "Increasing memory density by using KSM". In: *Proceedings of the Ottawa Linux Symposium*. Ottawa, Ontario, Canada, 2009, pp. 19–28 (Cited on page 47).

[AP01] G. Almasi and D. A. Padua. "MaJIC: A Matlab Just-In-Time Compiler". In: *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2001, pp. 68–81 (Cited on pages 27, 47).

[AST09] C. Ansótegui, M. Sellmann, and K. Tierney. "A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms". In: *Principles and Practice of Constraint Programming - CP 2009*. Springer, 2009, pp. 142–157 (Cited on page 16).

[BB12] J. Bergstra and Y. Bengio. "Random Search for Hyper-parameter Optimization". In: *The Journal of Machine Learning Research* 13 (2012), pp. 281–305 (Cited on pages 16, 97).

[BBH+14] B. Bischl, J. Bossek, D. Horn, and M. Lang. *Model-Based Optimization for mlr*. 2014. URL: `https://github.com/berndbischl/mlrMBO` (Cited on pages 9, 14, 93, 97, 104).

[BBS07] P. Balaprakash, M. Birattari, and T. Stützle. "Improvement Strategies for the F-Race Algorithm: Sampling Design and Iterative Refinement". In: *Hybrid Metaheuristics*. Springer, 2007, pp. 108–122 (Cited on page 16).

[BC84] R. A. Becker and J. M. Chambers. "Design of the S System for Data Analysis". In: *Communications of the ACM* 27.5 (05/1984), pp. 486–495 (Cited on page 19).

[BCF+09] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ICOOOLPS '09. Genova, Italy: ACM, 2009, pp. 18–25 (Cited on pages 27, 47).

[Ber18] A. Bertram. *Renjin: JVM-based Interpreter for the R Language for Statistical Computing*. 2018. URL: `http://www.renjin.org` (Cited on pages 3, 25, 27, 46).

[BG11] P. Bühlmann and S. van de Geer. *Statistics for High-Dimensional Data*. Springer, 2011 (Cited on page 1).

[BGS+08] P. Brazdil, C. Giraud-Carrier, C. Soares, and R. Vilalta. *Metalearning: Applications to Data Mining*. 1st ed. Springer, 2008 (Cited on page 16).

[BL18] K. Bache and M. Lichman. *UCI Machine Learning Repository*. 2018. URL: http://archive.ics.uci.edu/ml (Cited on pages 27, 29, 43).

[BLK+16] B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, and Z. M. Jones. "mlr: Machine Learning in R". In: *Journal of Machine Learning Research* 17.170 (2016), pp. 1–5 (Cited on pages 13, 31, 79, 97).

[BLM+15] B. Bischl, M. Lang, O. Mersmann, J. Rahnenführer, and C. Weihs. "BatchJobs and BatchExperiments: Abstraction Mechanisms for Using R in Batch Environments". In: *Journal of Statistical Computation and Simulation* 64.11 (2015), pp. 1–25 (Cited on page 97).

[Bor16] H. Borchers. *adagio: Discrete and Global Optimization Routines*. R package version 0.6.5. 2016. URL: https://CRAN.R-project.org/package=adagio (Cited on page 99).

[Bos16] J. Bossek. *smoof: Single and Multi-Objective Optimization Test Functions*. R package version 1.4. 2016. URL: https://CRAN.R-project.org/package=smoof (Cited on page 102).

[BRB+17] B. Bischl, J. Richter, J. Bossek, D. Horn, J. Thomas, and M. Lang. "mlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions". In: *arXiv preprint arXiv:1703.03373* (2017) (Cited on pages 14, 16).

[BWB+14] B. Bischl, S. Wessing, N. Bauer, K. Friedts, and C. Weihs. "MOI-MBO: Multiobjective Infill for Parallel Model-Based Optimization". In: *Learning and Intelligent Optimization Conference*. Florida, USA, 2014 (Cited on pages 4, 86, 88, 97, 101).

[CG13] C. Chevalier and D. Ginsbourger. "Fast Computation of the Multi-Points Expected Improvement with Applications in Batch Selection". In: *Learning and Intelligent Optimization*. Ed. by G. Nicosia and P. Pardalos. Springer, 2013, pp. 59–69 (Cited on page 88).

[Cha16] J. M. Chambers. *Extending R (Chapters 9 and 10)*. CRC Press, Taylor Francis Inc., 2016 (Cited on page 22).

[CJM12] M. Culp, K. Johnson, and G. Michailidis. *ada: an R Package for stochastic Boosting*. R package version 2.0-3. 2012. URL: http://CRAN.R-project.org/package=ada (Cited on page 29).

[CLC+17] W. Chang, R. Luraschi J., jQuery Contributors, M. Bostock, D. Contributors, and I. Sagalaev. *profviz: Interative Viszualizations for Profiling R Code*. R package version 0.3.4. 2017. URL: http://CRAN.R-project.org/package=profviz (Cited on page 28).

[CVB+02] O. Chapelle, V. Vapnik, O. Bousquet, and S. Mukherjee. "Choosing Multiple Parameters for Support Vector Machines". In: *Machine learning* 46.1-3 (2002), pp. 131–159 (Cited on pages 4, 14, 85).

[CWC+14] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao. "CMD: Classification-based Memory Deduplication Through Page Access Characteristics". In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 65–76 (Cited on page 47).

[DK14] C. Delimitrou and C. Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *ASPLOS '14*. Salt Lake City, Utah, USA: ACM, 2014, pp. 127–144 (Cited on page 93).

[DM98] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55 (Cited on page 74).

[DSH13] Y. Deng, L. Song, and X. Huang. "Evaluating Memory Compression and Deduplication". In: *Proceedings of the IEEE NAS '13*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 282–286 (Cited on page 47).

[Dyb09] R. K. Dybvig. *The Scheme Programming Language*. MIT Press, 2009 (Cited on page 19).

[F18] G. F. *TERR: TIBCO Enterprise Runtime for R*. 2018. URL: https://software.intel.com/en-us/mkl (Cited on page 19).

[FKH17] S. Falkner, A. Klein, and F. Hutter. "Combining Hyperband and Bayesian Optimization". In: Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS), Bayesian Optimization Workshop. 2017 (Cited on page 16).

[FSK08] A. Forrester, A. Soberster, and A. Keane. *Engineering Design via Surrogate Modelling: A Practical Guide*. Wiley, 2008, p. 288 (Cited on pages 17, 88).

[GAK+14] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. "Multi-Resource Packing for Cluster Schedulers". In: *SIGCOMM Computer Communication Review* 44.4 (08/2014), pp. 455–466 (Cited on page 93).

[GJL11] D. Ginsbourger, J. Janusevskis, and R. Le Riche. "Dealing with Asynchronicity in Parallel Gaussian Process based Global Optimization". In: *4th International Conference of the ERCIM WG on Computing and Statistics (ERCIM'11)*. 2011, pp. 1–27 (Cited on pages 5, 86, 91, 101).

[GLC10] D. Ginsbourger, R. Le Riche, and L. Carraro. “Kriging is Well-Suited to Parallelize Optimization”. In: *Computational Intelligence in Expensive Optimization Problems*. Springer, 2010, pp. 131–162 (Cited on pages 4, 19, 86, 88, 91, 94, 101, 115).

[GNU18] GNU libsigsegv: *GLibrary for Handling Page Faults in User Mode*. GNU Operating System, 2018. URL: http://www.gnu.org/software/libsigsegv/ (Cited on page 59).

[GU08] P. Grosjean and S. Urbanek. *R Benchmark 2.5 Suite*. 2008. URL: http://r.research.att.com/benchmarks/R-benchmark-25.R (Cited on page 60).

[GWH+13] J. Gareth, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning (Chapter 5)*. Springer, 2013 (Cited on page 15).

[HBZ09] K. Hornik, C. Buchta, and A. Zeileis. “Open-Source Machine Learning: R Meets Weka”. In: *Computational Statistics* 24.2 (2009), pp. 225–232 (Cited on page 13).

[HHL11] F. Hutter, H. H. Hoos, and K. Leyton-Brown. “Sequential Model-Based Optimization for General Algorithm Configuration”. In: *Learning and Intelligent Optimization*. Springer, 2011, pp. 507–523 (Cited on pages 2, 4, 14 sq., 18 sq., 101).

[HHL12] F. Hutter, H. H. Hoos, and K. Leyton-Brown. “Parallel Algorithm Configuration”. In: *Learning and Intelligent Optimization*. Springer, 2012, pp. 55–70 (Cited on pages 4, 19, 86, 88).

[HHZ06] T. Hothorn, K. Hornik, and A. Zeileis. *Unbiased Recursive Partitioning: A Conditional Inference Framework*. 3. R package version "1.0-13". 2006, pp. 651–674 (Cited on page 29).

[HVC16] R. T. Haftka, D. Villanueva, and A. Chaudhuri. “Parallel Surrogate-Assisted Global Optimization with Expensive Functions – A Survey”. In: *Structural and Multidisciplinary Optimization* 54.1 (07/2016), pp. 3–13 (Cited on pages 4, 87).

[IE94] I. o. E. IEEE and C. Electronics Engineers Inc. Staff. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX): System Application Program Interface (API), Amendment 1: Realtime Extension (C Language), IEEE Std 1003.1B-1993*. IEEE Standards Office, 1994 (Cited on page 74).

[IG96] R. Ihaka and R. Gentleman. “R: A Language for Data Analysis and Graphics”. In: *Journal of Computational and Graphical Statistics* 5.3 (1996), pp. 299–314 (Cited on pages 2, 19).

[Int18] Intel MKL: *Intel Math Kernel Library*. TIBCO, 2018. URL: https://docs.tibco.com/products/tibco-enterprise-runtime-for-r (Cited on page 27).

[JLG+12] J. Janusevskis, R. Le Riche, D. Ginsbourger, and R. Girdziusas. “Expected Improvements for the Asynchronous Parallel Global Optimization of Expensive Functions: Potentials and challenges”. In: *Learning and Intelligent Optimization*. Springer, 2012, pp. 413–418 (Cited on pages 5, 86, 91, 101).

[JLG11] J. Janusevskis, R. Le Riche, and D. Ginsbourger. *Parallel Expected Improvements for Global Optimization: Summary, Bounds and Speed-Up*. Tech. rep. 2011, pp. 1–21. URL: `https://hal.archives-ouvertes.fr/hal-00613971` (Cited on pages 5, 86, 101).

[Jon01] D. R. Jones. “A Taxonomy of Global Optimization Methods Based on Response Surfaces”. In: *Journal of Global Optimization* 21.4 (2001), pp. 345–383 (Cited on pages 16, 19).

[JSW98] D. R. Jones, M. Schonlau, and W. J. Welch. “Efficient Global Optimization of Expensive Black-Box Functions”. In: *Journal of Global Optimization* 13.4 (1998), pp. 455–492 (Cited on pages 4, 14, 17, 86).

[KBF+12] P. Koch, B. Bischl, O. Flasch, T. Bartz-Beielstein, C. Weihs, and W. Konen. “Tuning and Evolution of Support Vector Kernels”. In: *Evolutionary Intelligence* 5.3 (2012), pp. 153–170 (Cited on page 15).

[KE18] M. J. Kane and J. W. Emerson. *Not Quite R for the Parrot VM*. Yale University, 2018. URL: `https://github.com/NQRCore` (Cited on pages 3, 25, 27).

[KFB+17] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter. “Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets”. In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS 2017)*. Vol. 54. Proceedings of Machine Learning Research. PMLR, 2017, pp. 528–536 (Cited on page 16).

[KKE+14] H. Kotthaus, I. Korb, M. Engel, and P. Marwedel. “Dynamic Page Sharing Optimization for the R Language”. In: *Proceedings of the 10th Symposium on Dynamic Languages*. DLS ’14. Portland, Oregon, USA: ACM, 10/2014, pp. 79–90 (Cited on pages 8, 45).

[KKK+14] H. Kotthaus, I. Korb, M. Künne, and P. Marwedel. *Performance Analysis for R: Towards a Faster R Interpreter*. Abstract Booklet of the International R User Conference (UseR!) Los Angeles, USA, 07/2014 (Cited on pages 8, 25, 73).

[KKL+14] H. Kotthaus, I. Korb, M. Lang, B. Bischl, J. Rahnenführer, and P. Marwedel. “Runtime and Memory Consumption Analyses for Machine Learning R Programs”. In: *Journal of Statistical Computation and Simulation* 85.1 (2014), pp. 14–29 (Cited on pages 8, 25, 45, 71, 73).

[KKM15a] H. Kotthaus, I. Korb, and P. Marwedel. *Performance Analysis for Parallel R Programs: Towards Efficient Resource Utilization*. Tech. rep. 01/2015. SFB876 Project A3. Department of Computer Science 12, TU Dortmund University, 2015 (Cited on pages 8, 73).

[KKM15b] H. Kotthaus, I. Korb, and P. Marwedel. *Performance Analysis for Parallel R Programs: Towards Efficient Resource Utilization*. Abstract Booklet of the International R User Conference (UseR!) Aalborg, Denmark, 06/2015 (Cited on pages 8, 73).

[KKM16] I. Korb, H. Kotthaus, and P. Marwedel. "mmapcopy: Efficient Memory Footprint Reduction using Application Knowledge". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: ACM, 2016, pp. 1832–1837 (Cited on pages 8, 50, 71).

[KL18] H. Kotthaus and M. Lang. *BenchR: Set of Benchmark of R*. TU Dortmund University. 2018. URL: `https://github.com/allr/benchR` (Cited on pages 8, 29, 60).

[KLN+17] H. Kotthaus, A. Lang, O. Neugebauer, and P. Marwedel. *R goes Mobile: Efficient Scheduling for Parallel R Programs on Heterogeneous Embedded Systems*. Abstract Booklet of the International R User Conference (UseR!) Brussels, Belgium, 07/2017 (Cited on pages 9, 85, 112, 115).

[KMM+14] T. Kalibera, P. Maj, F. Morandat, and J. Vitek. "A Fast Abstract Syntax Tree Interpreter for R". In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 89–102 (Cited on pages 3, 25, 27, 46, 60).

[KN17] R. Koenker and P. Ng. *SparseM: A Sparse Matrix Package for R*. R package version 1.77. 2017. URL: `http://CRAN.R-project.org/package=SparseM` (Cited on page 45).

[KPM12] H. Kotthaus, S. Plazar, and P. Marwedel. *A JVM-based Compiler Strategy for the R Language*. Research Poster at The 8th International R User Conference (UseR!) WiP. Nashville, Tennessee, USA, 06/2012 (Cited on pages 8, 27).

[KRL+16] H. Kotthaus, J. Richter, A. Lang, M. Lang, and P. Marwedel. *Resource-Aware Scheduling Strategies for Parallel Machine Learning R Programs through RAMBO*. Abstract Booklet of the International R User Conference (UseR!) Stanford, USA, 06/2016 (Cited on pages 9, 85).

[KRL+17] H. Kotthaus, J. Richter, A. Lang, J. Thomas, B. Bischl, P. Marwedel, J. Rahnenführer, and M. Lang. "RAMBO: Resource-Aware Model-Based Optimization with Scheduling for Heterogeneous Runtimes and a Comparison with Asynchronous Model-Based Optimization". In:

Learning and Intelligent Optimization. LION 11. Vol. 10556. Lecture Notes in Computer Science. Springer, 2017, pp. 180–195 (Cited on pages 9, 85, 88, 99, 101).

[KSH+04] A. Karatzoglou, A. Smola, K. Hornik, and A. Zeileis. "kernlab - An S4 Package for Kernel Methods in R". In: *Journal of Statistical Software* 11.9 (2004). R package version 0.9-19, pp. 1–20 (Cited on page 29).

[Kt17] T. Kraljevic and H. team. *h2o: R Interface for H2O.* R package version 3.16.0.2. 2017. URL: `https://CRAN.R-project.org/package=h2o` (Cited on page 13).

[Kuh08] M. Kuhn. "Building Predictive Models in R Using the caret Package". In: *Journal of Statistical Software, Articles* 28.5 (2008), pp. 1–26. URL: `https://www.jstatsoft.org/v028/i05` (Cited on page 13).

[LBS17] M. Lang, B. Bischl, and D. Surmann. "batchtools: Tools for R to Work on Batch Systems". In: *The Journal of Open Source Software* 2.10 (2017) (Cited on page 103).

[LJD+17] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization". In: Proceedings of the International Conference on Learning Representations (ICLR). 2017 (Cited on page 16).

[LKM+15] M. Lang, H. Kotthaus, P. Marwedel, C. Weihs, J. Rahnenführer, and B. Bischl. "Automatic Model Selection for High-Dimensional Survival Analysis". In: *Journal of Statistical Computation and Simulation* 85.1 (2015), pp. 62–76 (Cited on pages 13, 16, 85).

[LSS12] I. Loshchilov, M. Schoenauer, and M. Sebag. "Self-adaptive Surrogate-assisted Covariance Matrix Adaptation Evolution Strategy". In: *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation.* GECCO '12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 321–328 (Cited on page 16).

[LW02] A. Liaw and M. Wiener. "Classification and Regression by randomForest". In: *R News* 2.3 (2002). R package version 4.6-7, pp. 18–22. URL: `http://CRAN.R-project.org/doc/Rnews/` (Cited on page 30).

[LYB+14] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition.* 1st. Addison-Wesley Professional, 2014 (Cited on page 27).

[Mar18] P. Marwedel. *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things (Chapter 6).* Third. Springer, 2018 (Cited on page 114).

[Mat60] B. Matérn. “Spatial Variation: Stochastic Models and their Application to some Problems in Forest Surveys and other Sampling Investigations”. In: *Meddelanden fran Statens Skogsforskningsinstitut* 49.5 (1960), p. 144 (Cited on page 104).

[MBC00] M. D. McKay, R. J. Beckman, and W. J. Conover. “A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code”. In: *Technometrics* 42.1 (2000), pp. 55–61 (Cited on pages 16, 97, 103, 116).

[MDH+12] D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, and F. Leisch. *e1071: Misc Functions of the Department of Statistics, Probability Theory Group, TU Wien*. R package version 1.6-2. 2012. URL: `http://CRAN.R-project.org/package=e1071` (Cited on page 29).

[MFR+13] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa. “XLH: More Effective Memory Deduplication Scanners Through Cross-layer Hints”. In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC’13. San Jose, CA: USENIX Association, 2013, pp. 279–290 (Cited on page 47).

[MHO+12] F. Morandat, B. Hill, L. Osvald, and J. Vitek. “Evaluating the design of the R language: objects and functions for data analysis”. In: *Proceedings of the 26th European conference on Object-Oriented Programming*. Beijing, China: Springer-Verlag, 2012, pp. 104–131 (Cited on pages 3, 8, 21, 27, 41, 45).

[Nea18] R. Neal. *pqR - a pretty quick implementation of the R programming language*. University of Toronto, 2018. URL: `https://github.com/radfordneal/pqR` (Cited on pages 3, 25, 27, 47).

[Neu17] O. Neugebauer. *Energy Measurement made Simple on Embedded Systems*. Technical Report No. 2 for Collaborative Research Center SFB 876 - Graduate School. 2017. URL: `https://sfb876.tu-dortmund.de/auto?self=%24egw1pio6bk` (Cited on pages 9, 115).

[NS17] F. Nan and V. Saligrama. “Adaptive Classification for Prediction Under a Budget”. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 4727–4737 (Cited on page 1).

[R C17a] R Core Team. *parallel: Support for Parallel Computation. R package first included in the R-core 2.14.0.* 2017. URL: `https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf` (Cited on pages 74 sqq., 79, 112 sq.).

[R C17b] R Core Team. *R Language Definition - Draft*. Version "3.4.3". 2017. URL: https://cran.r-project.org/doc/manuals/r-release/R-lang.pdf (Cited on pages 20 sqq.).

[R C18a] R Core Team. *An Introduction to R - Notes on R: A Programming Environment for Data Analysis and Graphics*. Version "3.4.4". 2018. URL: https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf (Cited on page 20).

[R C18b] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2018. URL: http://www.R-project.org (Cited on pages 3, 19, 45).

[R C18c] R Core Team. *R Internals*. Version "3.4.4". 2018. URL: https://cran.r-project.org/doc/manuals/r-release/R-ints.pdf (Cited on page 21).

[R C18d] R Core Team. *Writing R Extensions*. Version 3.4.4. 2018. URL: https://cran.r-project.org/doc/manuals/r-release/R-exts.html (Cited on pages 3, 20, 28).

[Rea12] Reactor Project: *Examining the R Language*. Purdue University, 2012. URL: http://r.cs.purdue.edu (Cited on page 28).

[RGD12] O. Roustant, D. Ginsbourger, and Y. Deville. "DiceKriging, DiceOptim: Two R packages for the Analysis of Computer Experiments by Kriging-based Metamodeling and Optimization". In: *Journal of Statistical Software* 51.1 (2012), pp. 1–55 (Cited on pages 19, 104).

[RIn18] RIncanter: *Use embedded R from Clojure and Incanter*. 2018. URL: https://github.com/jolby/rincanter (Cited on pages 3, 25).

[RKB+16] J. Richter, H. Kotthaus, B. Bischl, P. Marwedel, J. Rahnenführer, and M. Lang. "Faster Model-Based Optimization through Resource-Aware Scheduling Strategies". In: *Learning and Intelligent Optimization*. LION 10. Vol. 10079. Lecture Notes in Computer Science. Springer, 2016, pp. 267–273 (Cited on pages 9, 85, 88, 95).

[Ro13] G. Ridway and with contributions from others. *gbm: Generalized Boosted Regression Models*. R package version 2.1. 2013. URL: http://CRAN.R-project.org/package=gbm (Cited on page 29).

[RW05] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005 (Cited on page 17).

[SH13] K. Schliep and K. Hechenbichler. *kknn: Weighted k-Nearest Neighbors*. R package version 1.2-5. 2013. URL: http://CRAN.R-project.org/package=kknn (Cited on page 29).

[SK12] P. Sharma and P. Kulkarni. "Singleton: System-wide Page Deduplication in Virtual Environments". In: *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '12. Delft, The Netherlands: ACM, 2012, pp. 15–26 (Cited on page 47).

[SLA12] J. Snoek, H. Larochelle, and R. P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *NIPS workshop on Bayesian Optimization, Sequential Experimental Design, and Bandits*. 2012, pp. 2960–2968 (Cited on pages 16, 90 sq.).

[SWH+16] L. Stadler, A. Welc, C. Humer, and M. Jordan. "Optimizing R Language Execution via Aggressive Speculation". In: *Proceedings of the 12th Symposium on Dynamic Languages*. DLS 2016. Amsterdam, Netherlands: ACM, 2016, pp. 84–95 (Cited on pages 27, 46).

[SWM+89] J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn. "Design and Analysis of Computer Experiments". In: *Statistical Science* 4.4 (11/1989), pp. 409–423 (Cited on page 16).

[TAR13] T. Therneau, B. Atkinson, and B. Ripley. *rpart: Recursive Partitioning*. R package version 4.1-3. 2013. URL: `http://CRAN.R-project.org/package=rpart` (Cited on page 30).

[TDH12] J. Talbot, Z. DeVito, and P. Hanrahan. "Riposte: a trace-driven compiler and parallel VM for vector code in R". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. Minneapolis, Minnesota, USA: ACM, 2012, pp. 43–52 (Cited on pages 3, 25, 27, 46).

[TDH14] J. Talbot, Z. DeVito, and P. Hanrahan. "Just-in-time Length Specialization of Dynamic Vector Code". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: ACM, 2014, 20:20–20:25 (Cited on pages 27, 46).

[THH+13] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. "Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms". In: *Proceedings of ACM SIGKDD*. 2013, pp. 847–855 (Cited on pages 13, 85).

[Tie01] L. Tierney. "Compiling R: A Preliminary Report". In: *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*. 2001 (Cited on pages 20, 27).

[Tie18] L. Tierney. *Lisp-Stat*. 2018. URL: `http://homepage.cs.uiowa.edu/~luke/xls/xlsinfo` (Cited on pages 3, 25).

[TLB+15] M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell. "Resource-Aware Task Scheduling". In: *ACM Transactions on Embedded Computing Systems* 14.1 (2015), 5:1–5:25 (Cited on page 93).

[tra18] traceR: *Profiling Framework for the R Language*. TU Dortmund, 2018. URL: `https://github.com/allr/tracer` (Cited on pages 8 sq., 26, 28, 73, 75 sq.).

[TS13] K. Trapeznikov and V. Saligrama. "Supervised Sequential Classification Under Budget Constraints". In: *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*. Ed. by C. M. Carvalho and P. Ravikumar. Vol. 31. Proceedings of Machine Learning Research. Scottsdale, Arizona, USA: PMLR, 2013, pp. 581–589 (Cited on page 1).

[TTO+09] A. Tozawa, M. Tatsubori, T. Onodera, and Y. Minamide. "Copy-on-write in the PHP Language". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. Savannah, GA, USA: ACM, 2009, pp. 200–212 (Cited on page 48).

[VH04] S. Venkataraman and R. Haftka. "Structural Optimization Complexity: what has Moore's law done for us?" In: *Structural and Multidisciplinary Optimization* 28.6 (12/2004), pp. 375–387 (Cited on pages 4, 14 sq.).

[VPJ13] S. Valat, M. Pérache, and W. Jalby. "Introducing Kernel-level Page Reuse for High Performance Computing". In: *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. MSPC '13. Seattle, Washington: ACM, 2013, 3:1–3:9 (Cited on page 48).

[VR02] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Fourth. R package version 7.3-29. New York: Springer, 2002. URL: `http://www.stats.ox.ac.uk/pub/MASS4` (Cited on pages 29 sq.).

[WLL+05] C. Weihs, U. Ligges, K. Luebke, and N. Raabe. "klaR: Analyzing German Business Cycles". In: *Data Analysis and Decision Support*. Ed. by D. Baier, R. Decker, and L. Schmidt-Thieme. R package version 0.6-9. Springer-Verlag, 2005, pp. 335–343 (Cited on page 30).

[WPW15] H. Wang, D. Padua, and P. Wu. "Vectorization of Apply to Reduce Interpretation Overhead of R". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 400–415 (Cited on pages 27, 47).

[WWP14] H. Wang, P. Wu, and D. Padua. "Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization". In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '14. Orlando, Florida, 2014 (Cited on pages 3, 25, 27, 37, 47, 60).

[ZWW18] X. Zhang, Q. Wang, and S. Werner. *OpenBLAS: An Optimized BLAS Library*. 2018. URL: `http://www.openblas.net/` (Cited on pages 27, 74).

List of Figures

1.1 Simplified Visualization of the Model-Based Optimization Procedure. 4
1.2 Organization and Contributions of the Thesis. 7

2.1 Visualization of the mean missclassification error (mmce) as a performance measurement for different configurations of a SVM classification task with radial basis function kernel (SVM-RBF). 15
2.2 Visualization of the Sequential MBO (SMBO) Algorithm. 17
2.3 Visualization of two exemplary SMBO iterations with the *Lower-Confidence Bound* (LCB) as infill criterion. 18
2.4 Categories that influence and build up the R language environment. 20
2.5 Execution Model of the R Language. 23

3.1 Execution times for 10-fold cross validation of all machine learning algorithms, including model fit, prediction and calculation of the misclassification error on all selected datasets. Y axis is on $\log_{10}$ scale. 31
3.2 Categorization of the runtime optimization potential of R code. . 32
3.3 Runtime profiles relative to the total execution time of each benchmark. 34
3.4 Runtime profile categories of the R-Internal functions. 35
3.5 Runtime profile categories of the R-Provided functions. 37
3.6 Maximum memory usage versus total memory allocation for each benchmark. Y-axis is on log scale. 39
3.7 Memory profile categories of allocated R Data Structures. 40
3.8 Memory profiles relative to the total memory allocation of each benchmark. 40
3.9 Vector memory profile relative to the total memory allocation of vectors for each benchmark. 42

4.1 Example of the copy-on-write mechanism in the GNU R interpreter. R copies (duplicates) at object level instead of page level granularity. 49
4.2 Memory allocation scheme for dynamic page sharing strategies. . 50
4.3 Optimized object allocation via sharing a global zeroed page. . . 51
4.4 Optimized copy mechanism on page-level instead of object level granularity via page sharing. 53
4.5 Pseudo-code of a “full-overwrite” - where pages are not shared during allocation. 54
4.6 Deduplication optimization for zeroed pages. 55
4.7 Pseudo-code of the memory optimization core. 57

4.8 Page mapping and page fault handler. 58
4.9 Relative memory usage with page sharing optimization compared to standard R (lower is better). The 100% baseline represents the standard R interpreter (std. R) without optimizations. Geometric means for the memory savings are 13.6% for peak and 18.0% for average memory usage. 62
4.10 Memory consumption over time profiles for benchmarks with different memory behavior for the standard R interpreter vs. the interpreter with the page sharing optimization. Lines at the top indicate the peak memory usage, dotted lines mark the average memory usage. 65
4.11 Static refinement using different object size limits; Error bars have been omitted as the confidence intervals were smaller than 0.3% for all values (lower is better). 68
4.12 Exemplary memory consumption over time profile for the lssvm benchmark. Reaching a speed up by a factor of 5.2 on a system with 1GB of RAM. Lines indicate the peak memory usage, dotted lines mark the average memory usage. 70

5.1 Parallel execution mechanisms in R supported via packages. . . . 74
5.2 Relative CPU utilization and memory consumption of a parallel R program for calculating a Mandelbrot fractal on 4 cores on a single machine with prescheduling. 76
5.3 Relative CPU utilization and memory consumption of an R program for calculating a Mandelbrot fractal using 8 cores on 2 machines. 78
5.4 CPU utilization and memory consumption of an R program evaluating different parameter configurations of a SVM classification using 4 cores on a single machine with the default prescheduling mechanism. 80
5.5 CPU utilization and memory consumption of an R program evaluating different parameter configurations of a SVM classification using 4 cores on a single machine with the load balancing mechanism. 81
5.6 CPU utilization and memory consumption of an R program evaluating different parameter configurations of a SVM classification using load balancing on 4 cores on a single machine with 2 GB of main memory. 82

6.1 Runtime and mean misclassification error (mmce) for different configurations of an SVM classification. 86
6.2 Visualization of two exemplary MBO iterations with qLCB as infill criterion and two parallel configuration evaluations per iteration. 89

6.3 Exemplary scheduling for synchronous parallel MBO with q = 4 executed evaluations (jobs) per MBO iteration, with varying execution times leading to idling CPUs. 90

6.4 Exemplary scheduling for asynchronous parallel MBO to avoid CPU idling, with varying execution times that can lead to evaluations of similar configurations. 91

6.5 Resource-Aware Model-Based Optimization Framework. 93

6.6 Exemplary schedule of the resource-aware First Fit strategy with $q = 9$ jobs and $m = 3$ CPUs. 96

6.7 Averaged mean misclassification errors (MMCE): tuning error (left) and test data error (right) for the best observed configuration within a given time budget (lower is better). 98

6.8 Scheduling of MBO approaches: Runtime on x-axis and mapping of proposed candidates (gray boxes) to $m = 4$ CPUs on y-axis. Vertical lines indicate the end of a MBO iteration. Gaps represent idle time. 98

6.9 Exemplary schedule of the resource-aware knapsack based strategy with $q = 9$ jobs and $m = 3$ CPUs. 100

6.10 Visualization of the synthetic test functions for $d = 2$, used for the evaluation. `bohachevsky`(d) and `rosenbrock`(d) (upper part) show a smooth surface while `ackley`(d) and `rastrigin`(d) (lower part) are highly multimodal. 102

6.11 Averaged y value representing the best found configuration within a time budget of 2 h comparing the resource-aware scheduling strategies for the `rastrigin`(10) objective function with the `rosenbrock`(10) time functions on 16 CPUs (lower is better). 104

6.12 Residuals of the runtime estimation in the course of time for the `rosenbrock`(5) and `rastrigin`(5) time functions on 4 CPUs combined with `bohachevsky`(5) as objective function. Positive values indicate an overestimated runtime and negative values an underestimation. 105

6.13 Accuracy level vs. execution time for different objective functions using time function `rosenbrock`($\cdot$) (lower is better). 106

6.14 Scheduling of MBO algorithms. Time on x-axis and mapping of candidates to $m = 16$ CPUs on y-axis. Each gray box represents a job. Each red box represents overhead of the point proposal for the approaches. The gaps represent CPU idle time. 108

6.15 Accuracy level vs. execution time for different objective functions using time function `rastrigin`($\cdot$) (lower is better). 110

6.16 Accuracy level vs. execution time for the 2-dimensional objective functions using time function `rosenbrock`(2) (lower is better). . 117

6.17 Energy consumption in kJ on the two A15 CPUs (2.0 GHz), the two A7 CPUs (1.4 GHz) and combined consumption on both CPU types across all 10 repetitions for each objective function, with `rosenbrock`(2) time function and a time budget of 2 h (lower is better). 118

List of Tables

3.1 UCI classification tasks after pre-processing. The dataset ID, the number of observations and the number of features of the data sets are stored as numeric, integer or factor. 30
3.2 Mean misclassification rates over 10-fold cross validation for all input data sets. 31

4.1 Misc Benchmark Set. 60
4.2 Machine Learning Benchmark Set. 61
4.3 Memory Optimization Results: Std Peak - peak memory usage by the standard R interpreter; Opt Peak - peak memory usage by optimized interpreter; GainP - relative peak memory reduction achieved by optimization; Std Avg - average memory usage by the standard interpreter; Opt Avg - average memory usage by optimized interpreter; GainA - relative average memory reduction achieved by optimization; ZPG - number of zero pages found by the content check. 63
4.4 Runtime Results: Std - runtime with standard R interpreter; Opt - runtime with optimized R interpreter; Loss - relative runtime overhead incurred by optimized R interpreter; CC - number of executed content checks; Geometric mean of runtime loss is 5.3%; Confidence intervals have been omitted as they were smaller than 1.0% for all values. 67
4.5 Evaluation results with two configurations of RAM; see Table 4.3 and 4.4 for column descriptions, except Speedup: Runtime speedup factor (Std / Opt). Confidence intervals for runtime are shown (CI), others have been omitted as they are smaller than 0.8%. . 69

6.1 Ranking for accuracy levels 0.5, 0.1, 0.01 averaged over all benchmarks with `rosenbrock(·)` time function on 4 and 16 CPUs with a time budget of 4 h and 2 h, respectively. 107
6.2 Number of evaluated configurations per dimension across all 10 repetitions and objective functions with `rosenbrock(·)` time function on 4 and 16 CPUs with a time budget of 4 h and 2 h, respectively. 109
6.3 Ranking for accuracy levels 0.5, 0.1, 0.01 averaged over all problems with `rastrigin(·)` time function on 4 and 16 CPUs with a time budget of 4 h and 2 h, respectively. 111
6.4 Number of evaluated configurations per dimension across all 10 repetitions and objective functions with `rastrigin(·)` time function on 4 and 16 CPUs with a time budget of 4 h and 2 h, respectively. 111

6.5 Ranking for accuracy levels 0.5, 0.1, 0.01 averaged over all problems with `rosenbrock(2)` time function on 4 CPUs with a time budget of 2 h. 116
6.6 Number of evaluated configurations and performed model updates in brackets (for the synchronous MBO approaches), across all 10 repetitions for each objective functions with `rosenbrock(2)` time function on 4 CPUs with a time budget of 2 h. 117