Transferring Pareto Frontiers across Heterogeneous Hardware Environments

by

Pavel Valov

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

Supervisor:	Krzysztof Czarnecki, Professor David R. Cheriton School of Computer Science, University of Waterloo
Internal Members:	Peter van Beek, Professor David R. Cheriton School of Computer Science, University of Waterloo
	Grant Weddell, Associate Professor David R. Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Derek Rayside, Associate Professor Department of Electrical and Computer Engineering, University of Waterloo
External Examiner:	Jörg Andreas Kienzle, Associate Professor School of Computer Science, McGill University

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Configurable software systems provide options that affect functional and non-functional system properties. Selection of options forms system configurations with corresponding properties values. Nowadays configurable software systems are ubiquitously used by software developers, system administrators, and end users due to their inherent adaptability. However, this flexibility comes at a cost. Configuration space exponentially explodes with addition of new options, making exhaustive system analysis impossible.

Nevertheless, many use cases require a deep understanding of a configurable system behavior, especially if the system is redeployed across heterogeneous hardware. This leads to the following research questions: (1) Is it possible to transfer configurable software property information across a collection of heterogeneous hardware? (2) Is it possible to transfer relative configurations optimality information across heterogeneous hardware?

We address the first question by proposing an approach for transferring performance prediction models of configurable systems across heterogeneous hardware. This approach builds a predictor model to approximate performance on a source hardware, and a separate transferrer model to transfer approximated performance values to a destination hardware. Experiments on three configurable software systems across 23 heterogeneous hardware environments demonstrated high accuracy (less than 10% error) for all studied combinations.

We address the second question by proposing an approach for approximation and transferring of Pareto frontiers of optimal configurations (based on multiple system properties) across heterogeneous hardware. This approach builds an individual predictor and transferrer for each analyzed system property. Using trained models, we can build an approximated Pareto frontier on a source hardware and transfer this frontier to a destination hardware. Experiments on five configurable systems across 34 heterogeneous hardware demonstrated feasibility of the approach and that the accuracy of a transferred frontier mainly depends on the approximation rather than the transferring process, while being linearly proportional to a training sample size.

Acknowledgements

First of all, I would like to thank my supervisor Professor Krzysztof Czarnecki for this opportunity to perform research in University of Waterloo and for his guidance during my PhD studies.

Further, I would like to thank my research collaborators Jean-Christophe Petkovich and Dr. Jianmei Guo for their engineering and scientific knowledge, suggestions, and help in this project.

I would also like to thank Professor Derek Rayside, Professor Grant Weddell, Professor Jörg Andreas Kienzle, and Professor Peter van Beek for being my examining committee, for their reviews, feedback, and questions that helped me to expand and improve my work.

I would like to thank our Graduate Coordinator Paula Roser, for her patience, help, and constant willingness to help me in any situation.

I would like to express my sincere gratitude to my parents, my wife Alexandra, my sister Anna, and my best friend Eldar Khalilov. Thank you all for your unconditional support, patience, and advice through all these years. Without you this thesis would not be possible.

Finally, I would like to thank my friends from University of Waterloo who always supported me and made my studies much more joyful: Alex Murashkin, Changjian Li, Ed Zulkoski, Hemant Surale, Jesús Alejandro Padilla Gaeta, Hicham El Zein, Ivan Kobyzev, Leonardo Passos, Malek Naouach, Rafael Olaechea Velazco, Rodrigo Queiroz, Vyacheslav Derevyanko, and Zubair Akhtar Chaudhry.

Dedication

This work is dedicated to my parents, my wife Alexandra, and my sister Anna, people that I will always love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many modern software systems provide *configuration options*. These configuration options generally have a strong impact on a target system's functional behavior and non-functional properties, such as memory consumption, response time, and performance. Configuration options that are available to end users of a system are sometimes called *features* (Guo et al., 2013), while a specific selection of features' values determines a particular system *configuration*.

Nowadays highly configurable complex software systems, such as compilers, database engines, data compression software, and multimedia codecs, are more and more used by software developers and system end users. On the one hand, configurable software systems naturally allow to address various real-world functional and non-functional requirements due to their inherent adaptability and flexibility. Thus configurable software can conform to different use cases and reduce business expenses. But on the other hand, these benefits in applications of configurable software come at a cost. Exponential growth of the configuration space with introduction of new configurable features makes it virtually infeasible to fully and comprehensively analyze or test a subject system by exhaustively iterating through and measuring each possible system configuration.

Nevertheless, real-world business requirements might demand deep understanding of a system's functional and non-functional behavior. Moreover, many use cases require understanding of how a configurable system's behavior will change if the system is redeployed in a different hardware environment. Will a particular system configuration retain its' non-functional properties if the system is redeployed on a different hardware? Will a particular optimal configuration maintain it's relative efficiency when compared to other system configurations?

For example, similar questions may arise when a system administrator thoroughly analyzes a complex configurable software system on one hardware environment but needs to redeploy it to a different hardware. Thus the system administrator would be interested in understanding various system properties, like runtime performance and memory consumption, of the system across hardware, without investing much more effort into benchmarking and analysis of the system's configuration space. A resembling scenario might occur in the domain of software performance testing, when a software engineer wants to be sure that a configurable system meets minimal performance requirements across a range of different hardware platforms.

System performance by itself is considered to be one of the most important non-functional system properties, since it might directly affect business processes and the quality of interaction between

a business user and a system. Numerous and diverse research dedicated to performance prediction and modeling of software systems gave rise to an independent field of research called performance engineering. However, up until recently, not much attention was given to the aforementioned problem of transferring of performance knowledge across different hardware environments. Therefore, in the current work we specifically focus our research on performance of configurable software systems.

Unfortunately, previous research on prediction and transferring of performance of software systems doesn't always take into the account some restrictions that might occur in a real-world scenario (see Section 2.3 for details). For example, researchers investigate highly specialized hardware environments and software systems, expect a practitioner to have full control over the prediction and transferring process, etc. On the contrary, the main goal of our research is to develop a pragmatic methodology that could be applied in real-world scenarios. This goal resulted in several major constraints that guided the development of our approach.

First of all, we do not assume that a user has any control over the sampling process of a system's configuration space and might be restricted to historically benchmarked data only. Because of that, to explore systems' configuration spaces in our experiments we employ *pseudo-random sampling* in order to imitate this worst-case scenario of impossibility to make any sampling choices by a user.

Secondly, we assume that a practitioner might have limited information about actual system properties values, and it might not cover a configuration space entirely. Therefore, we analyze different sampling sizes in our experiments, in order to provide an assessment of how accurate predictors and transferrers might be.

Thirdly, we assume that a user can be working with a closed-source software and might not have any understanding about internal workings of a system, and won't be able to use this knowledge to improve predictors or transferrers. Consequently, our methodology regards analyzed software systems as 'black-boxes' that output particular properties' values, given a configuration and a workload.

Finally, our goal to make the approach practical imposes some restrictions on predictors and transferrers of studied system properties. Users should be able to: (1) train these models using minimal amount of training data, (2) build and validate the models in a completely automatic fashion, (3) visualize the models in an intuitive way, in order to get additional insights from training data and to verify that the models really work. All these requirements forced us to investigate basic machine learning methods as candidate models for our methodology, such as linear regression and regression tree models.

To sum up, described real-world challenges and requirements provide a research direction for our work. We can formulate a set of questions that provide a research framework for our study:

1. Is it possible to transfer information about configurable software system performance across a collection of heterogeneous hardware environments in a minimalistic and practical way?

2. Is it possible to transfer information about relative configurations optimality of a configurable software system performance across a collection of heterogeneous hardware environments in a minimalistic and practical way?

First of all, we begin our work by performing a comprehensive overview of related work that is the most similar to our own research, presented in Chapter 2. We focus on two major topics in

the literature review: (1) modeling and predicting performance of configurable software systems, and (2) transferring performance prediction models of highly configurable software systems across heterogeneous hardware environments. For each of the two topics we summarize the most relevant research and highlight the most important differences with our work.

Secondly, we address the first question about transferring performance information across heterogeneous hardware. We approach this question in Chapter 3 by transferring performance models of software systems. Previously, researchers have successfully demonstrated the correlation between feature selection and performance. However, the generality of these performance models across different hardware platforms has not yet been evaluated. We propose a technique for enhancing generality of performance models across different hardware environments using linear transformation. Empirical studies on three real-world software systems show that our approach is computationally efficient and can achieve high accuracy (less than 10% mean relative error) when predicting system performance across 23 different hardware platforms. Moreover, we investigate why the approach works by comparing performance distributions of systems and structure of performance models across different platforms.

Thirdly, we answer the second question about transferring relative configuration optimality information across heterogeneous hardware. We approach this question in Chapter 4 by utilizing a notion of Pareto optimality. We call a system configuration Pareto optimal if it is not possible to improve any of its properties values without worsening at least one other property value. A subset of configuration space forms a Pareto frontier of optimal configurations in terms of multiple properties, from which a user can choose the best configuration for a particular scenario. However, when a well-studied system is redeployed on a different hardware, information about property value and the Pareto frontier might not apply. We investigate whether it is possible to transfer this information across heterogeneous hardware environments.

We propose a methodology for approximating and transferring Pareto frontiers of configurable systems across different hardware environments. We approximate a Pareto frontier by training an individual predictor model for each system property, and by aggregating predictions of each property into an approximated frontier. We transfer the approximated frontier across hardware by training a transfer model for each property, by applying it to a respective predictor, and by combining transferred properties into a frontier.

We evaluate our approach by modeling Pareto frontiers as binary classifiers that separate all system configurations into optimal and non-optimal ones. Thus we can assess the quality of approximated and transferred frontiers using common statistical measures like sensitivity and specificity. We test our approach using five real-world software systems from the compression domain, while paying special attention to their performance. Evaluation results demonstrate that accuracy of approximated frontiers depends linearly on predictors' training sample sizes, whereas transferring introduces only minor additional error to a frontier even for small training sizes.

The thesis is partially based on two previously published papers. Chapter 3 is based on the paper 'Transferring Performance Prediction Models across Different Hardware Platforms' (Valov et al., 2017), authored by Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister and Krzysztof Czarnecki. Pavel Valov designed and implemented experiments with collected systems data, and wrote the paper itself. Jean-Christophe Petkovich designed and implemented benchmarks for the studied software systems and collected the necessary data by using the benchmarks on

DataMill cluster of heterogeneous hardware platforms. Jianmei Guo provided ideas, guidance, and reviews for the paper. Sebastian Fischmeister and Krzysztof Czarnecki provided general guidance for the paper.

Chapter 4 is based on the paper called 'Transferring Pareto Frontiers across Heterogeneous Hardware Environments' (Valov et al., 2020), authored by Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. Pavel Valov designed and implemented benchmarks for the studied software systems, collected system data using the benchmarks on Microsoft Azure cluster of heterogeneous hardware platforms, designed and implemented research experiments and wrote the paper itself. Jianmei Guo reviewed this work and provided feedback on methodology and writing of the paper. Krzysztof Czarnecki provided general guidance for the paper.

Chapter 2

Related Work

Performance engineering of software systems is a complex and diverse field of knowledge that encompasses a variety of approaches and techniques for performance analysis and prediction in different use cases (Woodside et al., 2007). In the current chapter we aim to review and highlight research that is the most relevant to our own work. Our research is primarily related to two major topics in performance engineering community: (1) model-based performance prediction of configurable software systems, and (2) transferring of performance prediction models across heterogeneous hardware environments. We discuss these topics separately in the following sections.

2.1 Software performance prediction

Performance prediction of software systems plays a major role in the domain of software performance engineering. Different aspects of this subdomain were covered in various literature reviews. Balsamo and Marzolla (2005) performed an in-depth review of white-box model-based performance prediction of software systems during a software development process. Koziolek (2010) carried out an exhaustive review of more than 20 methodologies for performance prediction of component-based software systems, that not only utilize classical performance modeling techniques, such as stochastic process algebras or queueing networks, but also employ a component-based structure of analyzed software systems. Abdelaziz et al. (2011) explore challenges of performance prediction for component-based software systems. Researchers analyze related work for three different approaches for performance prediction: model-based, measurement-based, and mixed approaches. Authors come to conclusion that the mixed approach for performance prediction is the most promising for building a comprehensive performance reasoning framework for component-based systems.

Although aforementioned surveys (Abdelaziz et al., 2011; Balsamo and Marzolla, 2005; Koziolek, 2010) provide a broad overview of different performance prediction methodologies, they virtually do not describe research that is relevant to our work. First of all, researchers mainly concentrate on white-box model-based prediction methods that are used during development phase of software systems. Conversely, we focus on black-box model-based prediction methods that usually require already implemented systems for analysis. Secondly, researchers mostly focus on monolithic, component-based, or distributed software systems, without concentrating on their particular

configurations. On the contrary, we focus on highly configurable software systems and analyze how their functional or non-functional properties change for different system configurations. Because of that, we perform our own literature review, highlighting the most relevant research and comparing it to our own work.

In the succeeding subsections we tried to group together and review research works that are the most relevant to black-box model-based performance prediction of highly configurable software systems. We classified all work into several main groups based on a type of configurable software that a particular work studies. We have selected this classification, since a type of a studied software usually has an impact on the overall process of performance prediction. We have formed four different groups based on this criteria: (1) general-purpose configurable software systems, (2) configurable solvers for propositional satisfiability and mixed integer problems, (3) configurable general-purpose planners for artificial intelligence problems, and (4) configurable high-performance algorithms for supercomputing hardware.

2.1.1 Performance prediction of configurable software systems

We begin with reviewing related work that deals with performance prediction of general-purpose configurable software systems, i.e. systems that can operate on Intel and AMD based hardware, and has a large user base, e.g. compilers, database engines, video codecs, etc. We understand that this kind of grouping is not exactly strict, but it allowed us to group together similar performance prediction methodologies.

Courtois and Woodside (2000) noticed that some performance-relevant relations, like resourse demand relations, have irregular and jagged shape. To approximate resourse demand relations by a simple analytical function, researchers employ multivariate adaptive regression splines (MARS, see Friedman, 1991) and extend them with two different heuristics: (1) for calculating confidence intervals of MARS predictions and (2) for sampling new data points to gain required accuracy with minimal amount of data possible. This research is similar to our own work, since researchers also implement a model-based approach by utilizing a regression model in order to interpolate a performance metric function that depends on a configuration of independent features. However, there are certain differences with our work. First of all, researchers use a different regression model than us – regression splines instead of regression trees (CART, see Breiman, 2017). Secondly, we focus on analysis and prediction of a total runtime under a certain benchmark as a system performance metric, while researchers concentrate on immediate CPU resource demands.

Siegmund et al. (2012a) propose a measurement-based quantitative performance prediction methodology for configurable software systems. The proposed methodology carries out performance prediction by making all performance-relevant influences and interactions explicit, thus potentially requiring a large amount of measured data. To lower the required amount of data for performance prediction, authors implement and compare several different sampling heuristics: feature-wise, pair-wise, triple-wise, and hot-spot. Feature-wise heuristic evaluates performance influence of each individual configurable feature by calculating a performance delta between two minimalistic configurations: one with the target feature turned on and another with the target feature turned off. Pair-wise heuristic measures an additional set of configurations in order to evaluate all possible pair-wise feature interactions, which are considered to be among the most widespread

performance-relevant interactions. Triple-wise heuristic allows to quickly detect interactions between three features, based on the assumption that if given three features provide a pair-wise interaction in any combination, then most likely these features provide a triple-wise interaction as well. Hot-spot heuristic detects highly-coupled features that significantly influence system performance. Using aforementioned heuristics it is possible to narrow down performance-relevant features and their interactions, calculate their performance deltas, and finally predict performance for a given configuration. The proposed approach was implemented in a tool called SPL Conqueror (Siegmund et al., 2012c), specifically designed for automatic performance measurement, prediction, and optimization. Although researchers also try to predict runtime performance of configurable software systems, they achieve it in a completely different fashion from ours. First of all, instead of using random sampling, researchers use aforementioned heuristics in order to select which feature interactions and system configurations to measure from the system's whole configuration space, what might not be possible in a real-world scenario where a practitioner might be limited by a fixed set of previously benchmarked data only. Secondly, instead of using machine learning techniques in order to figure out the relationship between system configurations and performance, researchers utilize a heuristic to approximate performance values for a particular configuration based on previously measured data.

Westermann et al. (2012) proposed an approach for automatic, measurement-based method for inferring performance prediction functions. To minimize the amount of measurements, they developed three algorithms that iteratively select new data points if necessary: random breakdown, adaptive random breakdown, and adaptive equidistant breakdown algorithms. To build actual performance prediction functions, the authors use four different regression and interpolation methods: MARS (Friedman, 1991), CART (Breiman, 2017), and GP (Rasmussen, 2004). To validate built prediction functions, they use three different strategies: random validation set, dynamic sector validation with local prediction error, and dynamic sector validation with global prediction error scope. They provide a framework for evaluation of function building methods for performance prediction and for evaluation of different combinations of function building methods and parameter tuning strategies. Finally, they evaluated the methodology for performance prediction in two industrial case studies. Despite the fact that researchers performed a comprehensive comparative study on performance prediction of configurable software systems, their work is different from ours. First of all, the main focus of our work is not to perform a comparison of different machine-learning methods or introduction of new sampling strategies, but to demonstrate that building and transferring of performance prediction models is possible. Secondly, the described methods rely on a higher level of control of a system's configuration space and sampling process by a practitioner, while our methodology doesn't have the same assumptions.

Guo et al. (2013) proposed a variability-aware approach for performance prediction of configurable software systems based on small random samples of measured configurations. To reveal a correlation between a selection of configuration options and system performance, the authors use CART (Breiman, 2017). They perform a case study of the proposed approach using six configurable software systems with different application domains, implementation languages, configuration spaces and sizes. This study shows that the proposed approach on average achieves prediction accuracy of 94%, when using small samples for CART prediction model training. Finally, the authors show that the approach achieves the best results when a training sample has a similar performance distribution as the whole population of configurations.

Valov et al. (2015) extended the work of Guo et al. (2013), by carrying out an empirical comparison of regression methods for the problem of variability-aware performance prediction. They compare prediction accuracy of four methods: CART (Breiman, 2017), Bagging (Breiman, 1996), Random Forest (Breiman, 2001), and Support Vector Regression (SVR, see Smola and Schölkopf, 2004). For each method the authors generate multiple parameter settings, by using Sobol sampling, and select parameters that provided the best, the average and the worst prediction accuracy for each method. By analysing prediction accuracy of methods for combinations of different parameter settings, target configurable software systems and training sampling sizes, they assess which methods provide the best prediction most of the time, i.e. which method is the most robust one. Results showed Bagging to be the most robust technique for performance prediction, even when allowing an interval for selecting the best performance accuracy.

Guo et al. (2018) extend their previous work (Guo et al., 2013) and propose a new data-efficient approach for performance prediction of configurable software systems. Authors provide an algorithm called DECART that combines regression trees (CART, see Breiman, 2017) with resampling and parameter tuning procedures in an automatic fashion. Researches also perform case studies of three resampling techniques and three parameter-tuning strategies. Finally, authors devise an analytical metric for assessing representation quality of a population by a given sample of measured configurations.

This cluster of work (Guo et al., 2013, 2018; Valov et al., 2015) is not only similar to our own research, but it also laid a foundation for our work. Researchers also employ regression models for performance prediction of configurable software and do it in a data-efficient way by using small samples of actually measured configurations. However, there are certain differences. First of all, researchers work with a wider range of regression models (see Valov et al., 2015). But the main difference with our work is that researchers completely do not investigate performance prediction and analysis across heterogeneous hardware platforms, what is the main focus of our work.

Sarkar et al. (2015) performed a thorough research in order to find the most optimal strategy for sampling in the domain of performance prediction of configurable software systems. Authors define optimality as a balance between effort of measuring performance of sampled configurations and between accuracy of performance prediction based on the sampled configurations. To assess quality of sampled configurations, authors estimate effort and accuracy together via a composite model of sampling cost. Researchers implement a novel heuristic, operating on configurable feature frequencies, for selecting a starting sample of training configurations. Authors improve projective sampling by injecting the implemented heuristic into the sampling process and by comparing the implemented heuristic with a classical t-way feature coverage based heuristic. Moreover, researchers not only compare different initializing heuristics, but also compare progressive and projective sampling techniques themselves in terms of the composite sampling cost. Finally, authors demonstrate via an empirical study that the most cost-effective sampling strategy is projective sampling technique, using exponential function as a projective function, with feature-frequency initialization heuristic. Although researchers also work on performance prediction of configurable software systems, this research is different from our work. Researchers concentrate on making this process more efficient by saving a measurement budget through prediction model quality assessment. We, on the contrary, focus on building and transferring performance prediction models and we do not assume that a practitioner might have any control over the measurement process.

Zhang et al. (2015) proposed a new approach for performance prediction of configurable software systems, which utilizes Fourier transform and can predict a particular configuration performance with required precision, while minimizing amount of actually measured configurations. Authors regard software systems' performance functions as Fourier sparse functions, since previous research (Guo et al., 2013; Siegmund et al., 2012b) has shown that these performance functions are not arbitrary but are instead structured, due to internal structure of the studied software systems. Based on this assumption, researchers devised a novel algorithm that is able to approximate these structured Fourier sparse functions using their Fourier decomposition, while satisfying user-specified accuracy and confidence levels requirements. Moreover, the proposed algorithm utilizes iterative and progressive random sampling, thus allowing to minimize the overall configuration measurement effort. Finally, authors perform a comprehensive evaluation of the proposed algorithm and compare it to other performance prediction techniques for configurable software systems. Zhang et al. (2016) continued their previous work (Zhang et al., 2015) by formulating a mathematical model for describing performance-relevant feature interactions (PRFIs) using Boolean functions. Researchers represent feature interactions as partial derivatives, where a first-order partial derivative represents a performance impact of a single configuration feature, while higher-order partial derivatives represent performance impacts of multiple features. Furthermore, authors propose two algorithms for automatic feature interactions detection that operate on small random samples of configurations, while possibly providing a specified confidence level and accuracy. Researchers expect the proposed mathematical formulation and discovery algorithms to apply not only to performance-relevant non-functional system properties, but to any quantifiable system characteristic. The main difference of this research by Zhang et al. (2015, 2016) with our work is the use of Fourier analysis for performance prediction.

Nair et al. (2018) proposed a novel approach called WHAT for performance prediction of configurable software systems using spectral learning. WHAT performs dimensionality reduction of configuration space by using a spectrum of a distance matrix between configurations. Thus it is possible to achieve more accurate and stable results with fewer sampled configurations. Authors claim that using two to ten times smaller samples of measured configurations the proposed approach achieves a comparable accuracy with smaller standard deviation compared to other state of the art techniques.

2.1.2 Performance prediction and optimization of complex algorithms

We continue by reviewing related research that works on performance prediction and optimization of complex algorithms for solving hard combinatorial, propositional satisfiability, and mixed integer programming problems. Although we do not study performance optimization in our own research, we highlighted some research from this domain, since it utilizes similar approaches and machine learning techniques.

Hutter et al. (2006) performed a research study in order to show that it is possible to utilize empirical hardness models to predict runtime performance of complex configurable algorithms. By empirical hardness models researchers understand machine learning models that are used to predict runtime performance of configurable algorithms in general and search algorithms in particular for this work. To implement actual empirical models, researchers use ridge regression (Hoerl and

Kennard, 1970). Unlike previous related work (Leyton-Brown et al., 2002; Nudelman et al., 2004) that concentrated on algorithms that are deterministic and complete, i.e. always provide a solution, researchers investigate incomplete randomized algorithms and specifically stochastic local search algorithm that is applicable to a variety of hard combinatorial problems. Moreover, researchers not only investigate harder practical problems, but also incorporate algorithm configuration options into empirical hardness models along with extracted features of a particular problem instance. Finally, researchers analyze whether it is possible to automatically tune algorithm parameters for each particular problem instance. This work is different from ours in several ways. First of all, researchers use different regression model instead of regression trees: ridge regression. Secondly, researchers focus on highly specialized software, stochastic local search algorithm, instead of general-purpose software systems.

Hutter et al. (2009) continue their work in performance engineering domain by concentrating on the problem of performance optimization of configurable algorithms. Researchers focus on model-based performance optimization of configurable randomized algorithms. Researchers select sequential parameter optimisation (SPO) based on Gaussian processes (Rasmussen, 2004) among several investigated optimization approaches as the most robust one. Authors thoroughly analyze the SPO approach and propose a new approach called SPO+ by extending SPO with a new intensification procedure and response values. Researchers validate the proposed approach based on two complex algorithms: (1) CMA-ES (Hansen and Kern, 2004), a derandomized evolution strategy (ES) with covariance matrix adaptation (CMA) for local search problem, and (2) SAPS (Hutter et al., 2002), an efficient dynamic local search algorithm. Finally researchers demonstrate that the new proposed approach achieves competitive performance when compared with other measurement-based optimization approaches.

Hutter et al. (2010b) continue their work on sequential parameter optimization of configurable algorithms' performance. Researchers introduce time bounds into Sequential Parameter Optimization (SPO) technique via random sampling of parameters, a novel intensification algorithm, and an updated prediction model. In the current work authors specifically focus on algorithms that have configuration parameters from continuous domains while using a single benchmark for each algorithm. Researchers concentrate on modifying SPO in order to provide solutions within specified time frames. First of all, researchers introduce interleaved random sampling of configuration parameter settings into the optimization process in order to skip expensive initial design. Secondly, authors develop a novel time-bounded algorithm to set a number of performed runs for each studied parameter setting, also called intensification algorithm. Thirdly, researchers make the optimization process even more efficient by replacing Gaussian process (Rasmussen, 2004) models with projected process models (Bottou et al., 2007; Quionero-candela et al., 2007; Rasmussen, 2004). Finally, researchers demonstrate that the proposed improvements to the sequential optimization provide major benefits, such as: much shorter times for parameter tuning, model building, and optimization process itself.

Hutter et al. (2011) continue their work on algorithm performance optimization via automatic parameter tuning. Researchers utilize and extend sequential model-based optimization technique (SMBO) that uses regression models to capture a relationship between a system's parameter configuration and a runtime performance. This technique alternates between a regression model fitting process and a process of selecting new configurations for measuring, what allows to use partially fitted models to select the most promising configurations. Researchers focus on making this method-

ology completely automatic and applicable to a wider range of configurable algorithms by allowing heterogeneous configuration parameters and by allowing optimization based on multiple domain problems. Authors introduce several improvements to SMBO such as: a modified intensification algorithm, random forest technique (Breiman, 2001) as a response surface, and a novel algorithm for selecting new configurations to be measured. Based on these improvements researchers introduce two SMBO-based optimization techniques: Random Online Aggressive Racing (ROAR) and Sequential Model-based Algorithm Configuration (SMAC). ROAR is an improved simplistic optimization technique based on a new intensification algorithm that uses uniform random sampling to select new configurations, while SMAC is improved even further and uses a model instead of random sampling for configuration selection. Authors selected several different configurable software systems to test the proposed optimization techniques: a local search SAT solver called SAPS (Hutter et al., 2002), a tree search SAT solver called SPEAR (Babić and Hutter, 2008), and an MIP solver CPLEX (CPLEX, 2009). Researchers evaluate the proposed optimization techniques on the studied systems and demonstrate that ROAR achieves comparable or better optimization performance than most of the competing optimizers and systems with exception of CPLEX, while SMAC improved optimization performance in most of the cases and never performed worse.

Although this cluster of work (Hutter et al., 2009, 2010b, 2011) mostly concentrates on performance optimization of configurable software, performance prediction is also used. However, there are certain differences with our work. First of all, researchers utilize performance prediction as an intermediate step in the overall performance optimization process in order to assess what is the most promising region of a configuration space to look for more performance-optimal configurations. Secondly, researchers use different regression models for performance prediction: Gaussian process, projected process, and random forest models.

Hutter et al. (2014, 2015) performed a comprehensive study of methods for configurable algorithms runtime prediction. The authors propose new methods for performance prediction based on random forests and approximate Gaussian processes. Moreover, the authors show how methods of survival analysis can be used for improving random forest technique to better handle incomplete performance measurements. With respect to the actual domain of algorithms which performance is predicted, the authors investigated satisfiability (SAT), travelling salesperson (TSP) and mixed integer programming (MIP) problems and inferred new probing and timing features for them. Finally, the authors present a comprehensive evaluation of different performance prediction methods including ridge regression and it's variants, neural networks, regression trees, Gaussian processes and random forests.

2.1.3 Performance prediction in AI domain

We continue our review by analyzing how performance prediction is applied in the domain of artificial intelligence problems. Research in AI domain, that we found to be the most similar to our own work, mostly concentrates on performance prediction of general-purpose AI planners in order to find the most efficient strategy for a particular AI problem solving or in order to come up with a more efficient general-purpose AI planner.

Hansen and Zilberstein (1996) propose a novel approach based on dynamic programming that combines fixed-time and online monitoring approaches in order to come up with a balanced planning

strategy for AI problem solving. The main focus of the paper is anytime algorithms, that have the ability to provide valid solutions at any time of their execution process, while providing the better solution the more time has passed since the start of the execution. This key ability of anytime algorithms allows practitioners to perform tradeoffs between the algorithm's execution time and the provided solution quality, but also requires an effort to predict when the solution quality is acceptable and the execution can be stopped. Problems in AI domain do not always exhibit determinism about the solution quality escalation speed and whether the time constraints will change after the start of the algorithm execution, which makes it impossible to use a classical fixed-time approach for determining total allowed algorithm running time. An alternative approach to select the most optimal execution time for an anytime algorithm is online monitoring of the algorithm's execution process, which allows higher agility and precision when working with tradeoffs, but requires a lot of additional continuous effort to perform the monitoring such as: (1) evaluating the already calculated solution, (2) assessing the probability of solution improvement, and (3) deciding whether to proceed with the process of solution improvement. In order to address the problem of high resource consumption by the online monitoring approach, researchers propose a new methodology that not only allows to find the most optimal time periods at which to monitor a running algorithm and to decide whether to continue it's execution or not, but also allows to answer: (1) how to select between the classical fixed runtime strategy and the online monitoring strategy depending on the performance variance of a studied algorithm, (2) how the monitoring periodicity should be correlated with the performance variance, and (3) how should the monitoring periodicity change with reaching an estimated end of runtime? Researchers address these targets by utilizing a dynamic programming approach in order to find not only the most optimal termination time, but also to decide whether to perform the monitoring itself, what allows to come up with a balanced monitoring strategy. Moreover, the proposed dynamic programming approach allows to assess tradeoffs between competing metrics, such as, monitoring cost and accuracy of solution quality estimation.

Howe et al. (1999) performed a comprehensive comparative study of general purpose AI planners which resulted in the development of a meta-planner BUS that provides better performance than any single planner through analysis and coordination of several planners. The goal of AI planning research is to implement the overall best-performing general purpose AI planner, however so far not a single planner has demonstrated a complete dominance not in the number and types of problems solved nor in solving time. Nevertheless, no comprehensive study had been performed on a wide range of heterogeneous planning problems and at least several planners in order to verify this 'no silver bullet' theory. Researchers tried to fill this gap by performing a systematical and comprehensive comparison of six open-source planners on more than 200 different benchmark planning problems. Empirical results of the study confirmed all major hypotheses that the authors proposed: (1) no single planner was able to demonstrate the best results for each and every benchmarking problem, (2) running time of planners depended on success of a particular planner when solving a given problem, since success and failure times vary greatly, and (3) overall performance of all planners highly depends on extractable features of a given planning problem and it's domain. Based on these observations, researchers developed a meta-planner called BUS that utilizes all six studied planners in order to achieve higher performance than any single one of them. In order to solve a given problem, BUS needs to decide in which order to try available planners on the problem. To do that, BUS builds two linear regression models for each planner based on empirical historical

data: one for predicting planner's running time and another for predicting whether a run will be successful. Based on these two linear models, BUS sets sequential execution order of all planners to increase probability of fast and successful solution to the given planning problem. Finally, BUS applies all solvers one-by-one until one of them successfully solves the problem or the time given to a particular solver elapses. Empirical evaluation showed that BUS completed more problems from the benchmark set than any other individual planner, while demonstrating smaller average runtimes than individual planners.

Roberts et al. (2007) continued the work of Howe et al. (1999) by performing a new study of planners in order to answer the following main questions: (1) whether it is possible to train high-quality planners' performance models using automatic machine learning methods, and (2) whether it is possible to use a trained performance model to enhance understanding of the respective planner and to even more improve it's performance. This new study is a comprehensive comparative research of general purpose AI planners and their performance that significantly enhances previous research and covers 28 different planners tested over 4726 various benchmarking planning problems. Authors demonstrate that it is possible to train highly accurate models of planners' probability of success and planners' running time based on historical data of solved planning problems, containing success rates and runtimes along with automatically extracted 32 feature values that characterize a problem and it's domain. For building success and runtime models, researchers tried and evaluated many different algorithms from the open-source data analysis software called WEKA: alternating decision tree (ADTree, see Freund and Mason, 1999), decision table majority classifier (DF, see Kohavi, 1995), Gaussian processes (GP, see MacKay, 1998), nearest-neighbour classifier with normalized Euclidean distance (IB1, see Aha et al., 1991), C4.5 decision tree (J48, see Quinlan, 1993), propositional rule learner RIPPER (JRip, see Cohen, 1995), instance-based classifier with entropic distance measure K* (see Cleary and Trigg, 1995), classification trees with leaves formed by logistic regression function, also called 'logistic model trees' (LMT, see Landwehr et al., 2005; Sumner et al., 2005), multinomial logistic regression with ridge estimator (Log, see Le Cessie and Van Houwelingen, 1992), multi-layer perceptron (MLP), and nearest-neighbor with non-nested generalized exemplars (NNge, see Martin, 1995; Roy, 2002). The authors perform a thorough evaluation and comparison of aforementioned machine learning models and highlight the ones that achieved the best results overall. The researchers propose a novel architecture for combining single planners into a structured portfolio in order to achieve a higher overall performance than any individual planner, by applying algorithm ranking and algorithm allocation techniques in order to improve scheduling of different planners and maximize overall efficiency of a portfolio. Through evaluation the researchers demonstrate that a correctly designed portfolio outperforms any individual general purpose planner. Finally, the authors discuss perspectives of how the performed comprehensive study can help to much deeper understand planners' performance nature and improve it's analysis and prediction.

The aforementioned research (Howe et al., 1999; Roberts et al., 2007) also focuses on runtime performance prediction of software systems, nevertheless, this research has several major differences with our work. First of all, the researchers specifically focus on the domain of general purpose AI planners, while we concentrate on general purpose software, thus trying to cover a larger user base. Secondly, the researchers perform a systematical comparison of a wide variety of different classification and regression models in order to engineer a novel scheduler of planners that on average outperforms any individual planner. On the contrary, we mostly concentrate on building

and transferring performance prediction models in the most practical and data-efficient way possible.

2.1.4 Performance prediction in supercomputing domain

So far we have discussed software systems that are executed on general-purpose computing hardware. In the current subsection we focus on high-performance supercomputing hardware platforms and corresponding software systems.

Lee et al. (2007) perform a comprehensive exploratory analysis and performance prediction of multiple configurable software systems across several hardware platforms. The researchers selected two configurable and highly parallel software systems: Semicoarsening Multigrid Algorithm (Brown et al., 2000) and High-Performance Linpack Benchmark (Davies et al., 2011); and three high-performance hardware platforms: BlueGene/L (Adiga et al., 2002), ALC (Braby et al., 2003), and MCR (Garlick and Dunlap, 2003). To get an insight of the selected systems' configuration spaces, the researchers use different machine learning techniques: (1) hierarchical clustering, for filtering out unnecessary features, (2) association analysis, to test existence of a particular feature-performance relation, and (3) correlation analysis to evaluate this relation. For building performance prediction models, the researchers utilize restricted cubic splines (Durrleman and Simon, 1989) and multi-layered fully-connected feedforward artificial neural networks (Haykin, 1994). This work shares a lot of similarities with ours. The researchers also utilize model-based machine learning techniques for predicting runtime performance of configurable software systems. The main difference with our work is that researchers concentrate on highly parallel configurable software systems for supercomputing hardware platforms, while we focus on classical general purpose configurable software applications for general purpose computing hardware. Moreover, the researchers utilize different regression models, splines and artificial neural networks, other than regression trees.

2.2 Cross-platform performance prediction

The performance engineering community has already provided a large body of research dedicated to cross-platform prediction of software systems. The researchers utilize different methods and techniques to acquire necessary data and implement the performance prediction process. In the current section we highlight work which is the most related to our method of transferring performance prediction models of configurable software systems across heterogeneous hardware environments.

Brewer (1994, 1995) performed a comprehensive study in order to implement and evaluate high-level libraries that are portable across a wide variety of highly parallel computers, like scalable multiprocessor systems or computer networks, while keeping near-optimal performance across these systems. High variability of different costs, such as communication and throughput, leads to a problem that there is no single data layout or algorithm that is always the most optimal for any parallel hardware platform. In order to address this problem, the researchers employ high-level libraries that contain a set of several coordinated parameterized implementations, what increases the probability of efficient operation across heterogeneous hardware platforms. Each library implementation has an associated linear model that is used to predict it's running time on a particular platform with respect to given parameters, such as problem size. This allows to automatically acquire the

best implementation and parameter configuration for a specified hardware platform and workload. The researchers implemented two high-level libraries to evaluate the proposed approach: a library for solving partial differential equations and another library for selecting the most optimal sorting algorithm for a particular hardware platform. The researchers evaluate the proposed approach using four different highly parallel systems: CM-5 (Hillis and Tucker, 1993), Intel Paragon (Esser and Knecht, 1993), MIT Alewife's simulator (Kubiatowicz and Agarwal, 2014), and the FORE ATM-based network of workstations (Thekkath et al., 1993). Empirical results demonstrate that the proposed approach selected the best implementation in more than 99% of cases on all studied platforms. Although the researchers also deal with cross-platform performance prediction of configurable software systems, their whole work and approach is completely different from ours. First of all, the researchers focus not on general-purpose computing hardware, but on highly parallel multiprocessor or distributed computing systems. Secondly, since the researchers work with highly specialized hardware, naturally they need corresponding software systems for their case studies. Because of that, the researchers did not just select, but completely implemented two high-level libraries with multiple internal parameterized implementations. This is completely different from our approach of selecting general purpose software systems in order to cover a larger user base. Finally, the researchers use cross-platform performance assessment not to transfer machine-learning-based performance prediction models, but to select the best-fitting internal implementation for a particular hardware platform.

Hoste et al. (2006) proposed an approach for performance prediction of a given software application on a set of hardware platforms, to find out which platform provides the best performance for the given application. The authors use a set of special applications, called a benchmark suite, that have their microarchitecture-independent characteristics collected and performance values measured across all studied hardware platforms. Using this analyzed benchmark suite, the authors build a data transformation matrix that is used to transform applications into points in so-called benchmark space using their microarchitecture-independent characteristics. The benchmark space is populated by applications from the benchmark suite and by the target application for which performance needs to be predicted. Finally, the performance prediction of the target application is carried out by taking a weighted average of the performance values of neighbouring applications, called proxies, in the benchmark space. Although the researchers also deal with performance prediction of software systems across different hardware platforms, this research is completely different from our own work. First of all, the researchers do not analyze different configurations of the studied software systems, what significantly simplifies the overall performance prediction process. Secondly, the proposed approach relies on extensive upfront measurements of platform-independent characteristics and performance values of multiple benchmark-applications over various hardware environments. On the contrary, our methodology tries to be as data-efficient as possible and doesn't require measurements of additional benchmark-applications. Finally, as stated by the researchers themselves, if a studied software system doesn't have any neighbours in the benchmark-space, accurate performance prediction might not be possible, since there would be no neighbouring performance values to average upon, while our approach doesn't have such restrictions.

Thereska et al. (2010a,b) proposed an approach for performance modelling of complex popular applications such as Microsoft's Office suite and Visual Studio. All explored applications were specifically instrumented to export their current state as well as all necessary performance relevant metrics. Moreover, all explored applications were deployed on multiple machines, thus allowing

them to monitor how each particular application with different configurations behaves in various hardware platforms. To predict performance of a system on a particular hardware platform, the authors (1) select configuration options (both software and hardware) that influences a chosen performance metric the most, (2) use similarity search to select hardware platforms with similar hardware configuration, (3) return a distribution of possible performance metric values from a number of similar configurations as the result. Although the researchers also work on the problem of performance prediction of configurable software systems across heterogeneous hardware environments, they approach this problem in a completely different fashion. First of all, the proposed approach requires a thorough instrumentation of the studied software system in order to collect performance information for different software configurations and states, which requires additional development effort and some knowledge about an application's structure, what makes this approach not entirely black-box. Secondly, as stated by the researchers themselves, the proposed approach requires (1) an expensive hardware infrastructure with a cost ranging from several thousands to more than a hundred thousands of dollars and (2) a team of tens of maintainers to support this infrastructure. In general, the proposed approach tries to collect an exhaustive amount of data about the studied applications including their different configurations and states on a maximum amount of hardware platforms, while relying on a thorough instrumentation, hardware infrastructure, maintenance team, and extensive funding. On the contrary, our approach is black-box, minimalistic, and doesn't require additional infrastructure, team or funding.

Jamshidi et al. (2017a) perform a thorough exploratory analysis of the problem of transfer learning for performance prediction of configurable software systems across different environmental conditions, such as heterogeneous hardware, varying workload, and differing software versions. The authors aim not only to perform an actual transfer learning, but to understand why does the proposed approach work in principle and in which circumstances does it work. The researchers formulate four major research questions for their work, and several hypotheses for each question in order to develop it. First of all, the researchers try to analyze how does the performance behavior of the studied systems change across different hardware environments. By using Pearson linear correlation (Bishop, 2006), the authors demonstrate that it is possible to apply linear transformation between two hardware environments in order to transfer performance models when there are only minor hardware differences between these environments. By using Kullback-Leibler (KL) divergence (Cover and Thomas, 1991), the authors show that a software system, deployed on hardware environments with very strong differences, might have very similar distributions on these environments, what doesn't guarantee a possibility of a linear transformation between these environments, but might indicate that a more complex non-linear transformation is applicable. By comparing the 10th percentile of top and bottom configurations by performance on different hardware environments, the authors demonstrate that configurations generally retain their relative performance positions across varying hardware. Secondly, the researchers try to analyze whether configuration features of studied software systems retain their performance influence across different hardware environments. By using a paired t-test (Bishop, 2006), the authors show that only a subset of configurable features has any impact on the system performance and this impact is generally preserved across different hardware. By using regression trees (Breiman, 2017), the authors demonstrate that the relative strength of influence of configuration features on system performance generally remains the same across varying hardware. Thirdly, the researchers try to analyze if performance-relevant feature interactions are retained by configurable software systems across different hardware. By

using step-wise linear regression models (Siegmund et al., 2015), the authors demonstrate that although the majority of feature interactions remain across different hardware and retain their impact on performance, only very few feature interactions actually have any effect on a system performance in the first place. Fourthly, the researchers try to analyze whether configurations that do not lead to a successful execution of a benchmark, and are in fact invalid, stay invalid across different hardware. By calculating a percentage of identical configurations that are invalid across hardware environments, the authors demonstrate that such configurations are common and generally preserve their status across platforms. By using multinomial logistic regression (Bishop, 2006), the authors show that it is possible to train a model to predict whether a particular configuration is valid or not and transfer this knowledge across different hardware, which allows to exclude whole regions of the configuration space from analysis. This work is similar to ours in many ways. We also demonstrate that: (1) it is possible to employ linear transformation to transfer performance prediction models across different hardware, (2) by visualizing structure of regression trees across hardware platforms, that only a subset of configurable features influence performance on all platforms. Moreover, our work (Valov et al., 2017) was published earlier and is cited by Jamshidi et al. (2017a). However, the researchers perform a more comprehensive analysis in many ways: (1) by performing cross-platform performance distribution analysis, (2) by comparing relative performance of configurations and relative performance influence of features across hardware, (3) by analyzing performance influence and consistency of feature interactions across platforms, (4) and by cross-platform analysis of invalid configurations.

Jamshidi et al. (2017b) propose a new approach for improving performance prediction of highly configurable systems in a self-adaptation context. The researchers investigate the configuration space of a studied system at a lower cost, by examining not the actual system itself, but a proxy of the system, e.g., a simulator of a robotic device. Then, it is possible to approximate the relationship between the proxy and the system by utilizing a regression model based on a small sample taken from the actual system. Moreover, the researchers introduce a cost model that takes into account not only the model accuracy, but also the measurement effort needed to acquire necessary learning data. Thus, a practitioner can select a Pareto-optimal learning strategy, based on the available measurement budget and required accuracy. Although the researchers also deal with transferring performance prediction knowledge, they do it in a completely different fashion from ours. First of all, the researchers transfer knowledge between a simulator, running on a general-purpose hardware, and an actual hardware robotic system, while we transfer performance prediction models across general-purpose hardware and cloud-based hardware systems. Secondly, the researchers utilize Gaussian processes (Rasmussen, 2004) for both performance prediction and performance transferring models, while we use regression trees Breiman (2017) and linear models instead.

2.3 Summary

Finally, after analyzing the most relevant work one by one, we summarize our analysis and highlight the most important differences with our work. We group these differences by several major topics: (1) *hardware environments* used for deploying configurable software, (2) studied *configurable software systems*, (3) *sampling techniques* used for investigating a software systems' configuration spaces, (4) a *metric* used for measuring system performance, (5) *prediction models* used for assess-

ing a software systems' performance, and (6) *transferring models* used for transferring performance knowledge across different hardware environments.

We specifically selected *general purpose computing hardware*, based on Intel or AMD central processing units, for our case studies, in order to make our methodology reproducible, compatible with a wide variety of software, and thus automatically applicable to a large user base. Nevertheless, a large body of related research is dedicated to platforms from highly parallel and supercomputing domains: BlueGene/L, ALC, MCR (used in Lee et al., 2007), CM-5, Intel Paragon, MIT Alewife's simulator, FORE ATM-based network of workstations (used in Brewer, 1994, 1995). Moreover, some studies are devoted to robotic hardware systems, such as work by Jamshidi et al. (2017b).

As for software itself, we again focus on *general-purpose configurable software systems* that can perform on Intel and AMD based hardware, and have a large user base, thus making our approach more reproducible and practical. However, a large amount of related work analyze highly specialized software such as: AI planners (see Howe et al., 1999; Roberts et al., 2007), different kinds of SAT solvers (SAPS, SPEAR, see Hutter et al., 2011), various kinds of MIP solvers (CPLEX, see Hutter et al., 2011), SPEC CPU2000 benchmarks (see Hoste et al., 2006), local search algorithms (CMA-ES, see Hutter et al., 2009), or even specifically implemented custom software for solving complex engineering and scientific problems on highly parallel supercomputing hardware (see Brewer, 1994, 1995). Moreover, some research, like (Hoste et al., 2006), doesn't consider different system configurations, which substantially simplifies the performance prediction problem.

We try to make our methodology as practical as possible, because of that we assume that a practitioner might not have any kind of control over a sampling process and might be restricted to previously measured data only. Therefore, we use *pseudo-random sampling* to sample configuration spaces of studied software systems, in order to mimic this worst-case practical scenario. However, the related work does not generally have this assumption and the researchers use a variety of different techniques to sample the configuration spaces of systems, such as: exhaustive feature-wise, pair-wise, and triple-wise sampling heuristics (used in Siegmund et al., 2012a), random breakdown, adaptive random breakdown, and adaptive equidistant breakdown algorithms (used in Westermann et al., 2012), or generally assume full control by a practitioner of the sampling process (Sarkar et al., 2015). Moreover, some studies require comprehensive measurements of microarchitecture-independent characteristics across a collection of hardware platforms in advance (Hoste et al., 2006).

We use a *total running time to complete a specified benchmark* as a metric to assess performance of a given configurable software system. We chose this metric since we consider it to be more practically useful, intuitive, and easy-to-implement than other metrics used in the related work such as immediate CPU execution demand (Courtois and Woodside, 2000).

We use *regression trees* (Breiman, 2017) as a model for capturing relations between system configurations and system's performance metric values, as well as a model for capturing relations between metric values across different hardware environments. We chose regression tree since: (1) it can build non-trivial regression models based on a tiny amount of data (at least two non-identical observations) what is crucial for our research, since we want our methodology to work based on small training samples, (2) it has a small training time even on large datasets, (3) it has an intuitive binary tree structure that can be easily visualized and understood by a practitioner, (4) it can be easily transformed to a set of simple rules and reimplemented in any other programming language or modelling system, and finally (5) it can successfully capture complex feature interactions that

might lead to unexpected system performance anomalies. All these qualities make regression tree a perfect model for our methodology that tries to be minimalistic and practical. However, related work generally uses different regression models, since their premises and requirements are usually different. Nevertheless, we plan to investigate the applicability of these models to our problem in future work: alternating decision tree (ADTree, used in Roberts et al., 2007), bagging of regression trees (used in Valov et al., 2015), C4.5 decision tree (J48, used in Roberts et al., 2007), Gaussian Processes (GP, used in Hutter et al., 2009, 2010b; Jamshidi et al., 2017b; Roberts et al., 2007; Westermann et al., 2012), Fourier analysis (Zhang et al., 2015, 2016), logistic model trees (LMT, used in Roberts et al., 2007), multilayered fully-connected feedforward artificial neural networks (used in Lee et al., 2007), multilayer perceptron (MLP, used in Lee et al., 2007), multinomial logistic regression with ridge estimator (Log, used in Roberts et al., 2007), multivariate adaptive regression splines (MARS, used in Courtois and Woodside, 2000; Westermann et al., 2012), projected process (used in Hutter et al., 2010b), random forest (used in Hutter et al., 2011; Valov et al., 2015), restricted cubic splines (used in Lee et al., 2007), ridge regression (used in Hutter et al., 2006), and support vector regression (SVR, used in Valov et al., 2015).

We also use *linear transformation* as a model for transferring performance prediction knowledge across different hardware environments. We use linear regression because: (1) as well as a regression tree, a linear regression can build non-trivial models based on at least two measured configurations with different performance values, what allows to build and transfer a performance prediction model across two hardware environments by using only four non-identical measurements in total (two measured configurations on a source hardware platform to build a non-trivial regression tree and two more measurements of the same configurations on a target hardware platform to build a linear transformation model), (2) it also performs very fast on a massive amout of data, (3) it is one of the most studied and intuitive models and can be easily visualized by a practitioner in order to gain a better understanding of the transferring process. All these properties make linear transformation a perfect model for our minimalistic and practical approach. Nevertheless, other researchers use different regression models and techniques to perform knowledge transfer across hardware platforms like: Gaussian processes (used in Jamshidi et al., 2017a), and custom transferring techniques that require either extensive instrumentation of a studied software system, additional hardware infrastructure, and maintenance team, (Thereska et al., 2010a,b), or extensive upfront performance measurements of multiple benchmark software instrances across a variety of hardware platforms (Hoste et al., 2006). In our future work, we also plan to try the Gaussian processes model (Rasmussen, 2004), since it has demonstrated good results for the performance knowledge transferring problem, and MARS (Friedman, 1991), since we believe that regression splines might provide a very good fit for our transferring data.

We try to make our approach for transferring performance prediction models and Pareto frontiers of optimal configurations across heterogeneous hardware platforms as practical as possible, what affects each and every aspect of the proposed methodology. We performed our experiments using generally available hardware platforms and open-source configurable software systems. We used pseudo-random sampling for software configuration space exploration in order to imitate a scenario, when a practitioner doesn't have any control over the sampling process and must rely on previously collected data only. We use regression trees and linear regression as our prediction and transferring models respectively, because of their ability to work with a minimal amount of data, simplicity of visualization, and understandability by a practitioner. Moreover, our approach is completely

black-box and doesn't rely in any way on the knowledge about internal structure of the studied configurable software systems. We are not aware of any other work that combines all these properties for the problem of building and transferring of performance prediction models and Pareto frontiers.

Chapter 3

Transferring Prediction Models

Many software systems provide *configuration options*. These configuration options usually have a direct influence on the functional behavior of the target software systems. Some configuration options may impact systems' non-functional properties, such as response time, memory consumption and throughput. Configuration options that are relevant to users are usually called *features* (Guo et al., 2013), and a particular selection of features defines a system *configuration*.

Performance prediction of configurable software systems is a highly-researched topic (see Chapter 2 for details). For example, Guo et al. (2013) predicted a system configuration's performance by using regression trees based on small random samples of measured configurations. However, the majority of the previous related work analyzed studied configurable software systems based on measurements from a single hardware environment only and didn't analyze whether or not it is possible to transfer performance prediction models for configurable software systems across different hardware platforms.

The need for transferring performance prediction models occurs in many application scenarios. For example, a user of a software system performs a thorough performance benchmarking of the system and builds a performance prediction model for it. However, the prediction model is built only for the particular benchmarked machine. The performance prediction process on a different machine may not be able to directly reuse previous benchmarking results and prediction models. Modern Software as a Service (SaaS), Platform as a Service (PaaS) and other cloud-based industries face similar challenges. Based on historical performance data collected for their software on one cluster, users want to know how to tune the performance of their software systems for a new cluster with a different hardware, or how to select the best hardware platform with which to build their cluster.

In the current chapter we investigate the problem of performance prediction model transfer across different hardware environments. We make the following contributions:

- We propose an approach for transferring performance models of configurable software systems across platforms with different hardware settings. This approach (1) builds a performance prediction model based on a small random sample of configurations measured on one hardware platform and (2) transfers this model using linear transformation to other hardware platforms.

- We implement the proposed approach and demonstrate its generality using three real-world configurable software systems. Our empirical results show that for the majority of model transfers our approach achieves a high prediction accuracy (less than 10% mean relative error). We also observe a decreasing trend of mean relative error with the increase of the training data for the performance prediction model and for the linear transformation model.
- We carry out a thorough exploratory analysis to understand why our approach works. We compare performance distributions and structure of performance prediction models across different hardware platforms and show that the more similar distributions and prediction models across different platforms, the better transfer results are. Moreover, we carry out a comparative analysis of our methodology for different configurable systems and assess the time costs of our method.

Source code and data to reproduce our experiments in the current chapter are available online at https://bitbucket.org/valovp/icpe2017.

3.1 Motivating Example and Notation

Our objective is to enable the transfer of performance prediction results from one hardware platform to another. Consider purchasing a new hardware platform to encode large amounts of video using x264, which is a configurable application for encoding video streams in the H.264/MPEG-4 AVC compression format. Media encoding programs such as x264 usually have a large number of configurable features; tuning them has a significant impact on the quality of the video output and on the time necessary to encode it. In our example, we have measurements for 11 different configuration features, each with 2 individual settings. Obtaining these performance measurements with a video that takes a modest 15 minutes to encode requires 1536 hours of execution time. Rather than exhaustively measuring the same configurations on a new unstudied platform, it would be better to reuse performance data from previous tuning experiments to predict the performance of configurations on this new platform.

To formalize the problem of performance prediction, we represent features of a configurable software system as a set of binary decision variables $F = \{f_1, f_2, \dots f_{N_f}\}$, where f_i represents a particular variable and N_f represents the total number of features of the configurable software system. Each configuration $\mathbf{c}$ of the system is a set of value assignments to N_f variables f_i. We denote the set of all valid configurations of the system by $\mathbf{C}$. Table 3.1 represents a sample of 10 configurations along with their measured performance values. Each row of the Table 3.1 represents a particular configuration of x264, while each column represents a particular feature of the system.

We define the performance of a system as the total time required to execute a particular system benchmark. Performance of each configuration is expected to differ in a heterogeneous collection of machines that we denote by $M = \{m_1, m_2, \dots, m_{N_m}\}$, where m_i represents a particular machine or hardware platform and N_m represents the total number of machines in the collection. Each valid configuration, $\mathbf{c}$, of the software system has an actual performance value, $a_{\mathbf{c},m_i}$, on a machine m_i, a set of configurations, $\mathbf{C}$, has a set of actual performance values, $A_{\mathbf{C},m_i}$, on machine m_i. We define *training machine*, m_{trn}, as a machine that is used to build performance prediction models

for a given configurable software system (e.g., x264). In a practical setting, the training machine is one on which a particular software system is already well-studied and historical performance data for the system is acquired. We define *target machine*, m_{tgt}, as a machine to which performance prediction models must be transferred.

For example, we acquire a small random sample of configurations, $\mathbf{C}_S \subset \mathbf{C}$, along with their actual performance values, $A_{S,m_{trn}} \subset A_{\mathbf{C},m_{trn}}$, together forming sample, $S_{m_{trn}}$, on our training machine m_{trn}. Our goal is to predict the performance of all other configurations, $\mathbf{C} \backslash \mathbf{C}_S$, on machine m_{trn} based on this small random sample $S_{m_{trn}}$, and subsequently to predict the performance of the whole set of valid configurations $\mathbf{C}$ on all other machines in the collection M.

Table 3.1: Sample of 11 randomly-selected configurations of x264 system, along with their actual performance measurements on Machine №75

Conf.	Features											Perf. (s)
$\mathbf{c}_i$	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	$a_{\mathbf{c_i},m_{75}}$
$\mathbf{c}_1$	0	0	1	1	1	0	1	0	0	0	1	52.01
$\mathbf{c}_2$	0	1	0	1	1	1	0	0	1	0	0	24.09
$\mathbf{c}_3$	1	0	0	0	1	1	0	0	0	0	1	58.13
$\mathbf{c}_4$	1	1	0	1	1	0	0	1	0	1	0	37.49
$\mathbf{c}_5$	0	0	1	0	1	1	0	0	0	0	1	75.89
$\mathbf{c}_6$	1	1	0	1	0	0	0	1	0	0	1	51.05
$\mathbf{c}_7$	1	1	1	0	0	1	0	0	0	0	1	82.15
$\mathbf{c}_8$	1	0	0	1	1	1	0	0	0	0	1	41.40
$\mathbf{c}_9$	1	0	0	1	0	0	0	1	1	0	0	23.16
$\mathbf{c}_{10}$	0	0	0	0	1	0	1	0	1	0	0	23.20
$\mathbf{c}_{11}$	0	0	1	0	1	0	1	0	1	0	0	28.95

3.2 Transferring Performance Prediction Models

The overall process of transferring performance prediction models across different hardware platforms is sequential and can be separated into the following main steps: (1) training the performance prediction model, (2) training the linear transfer model, and (3) transferring the prediction results.

3.2.1 Training the Performance Prediction Model

We used regression trees for building models of the performance effects of software features. We selected regression trees as our method of model construction as they have been extensively used for performance prediction of configurable software systems and demonstrated good results (Guo et al., 2013; Thereska et al., 2010a,b; Valov et al., 2015; Westermann et al., 2012). The resulting prediction models can be graphically represented and readily understood by end users. Regression trees proved effective for a thorough exploratory analysis for our model transferring problem (see Section 3.3.5 for more detail).

We need to choose a method for sampling training data to build regression trees. Previous studies of feature performance modelling, Guo et al. (2013) and Valov et al. (2015), used small random samples of measured configurations to build prediction models. The use of random sampling in those works was motivated by the idea that in practice, available measured configurations of a system might not follow any particular feature-coverage criteria and would be essentially random. In our study we also use small random samples for prediction model building.

For the purpose of experiment reproducibility we provide datasets of measured configurations for each studied software system (for more details see Section 3.3.1). However, measuring the entire configuration space $\mathbf{C}$ of each software system under test was prohibitively expensive and couldn't be done in the time budget available (for more details see Section 3.3.1). Therefore, for each system we measured only a subset of valid configurations $\mathbf{C}_{exp} \subset \mathbf{C}$.

To choose which configurations should be included in $\mathbf{C}_{exp}$ we used experimental design techniques. Experimental design is an efficient procedure for obtaining experimental data that can be analysed to produce valid results (Croarkin and Tobias, 2013). Experimental design techniques can maximize information obtained by a practitioner for a given experimentation budget. Selection of a concrete design for a particular experiment depends on the goal of the experiment and the number of variables involved.

In our study we use "screening" experimental designs, where the goal is to "screen out" or select the main effects that influence the response variable (in our case system performance), such as *full factorial* and *fractional factorial* experimental designs. This is achieved by selecting the most "informative" configurations for a given system.

Full factorial design generates all possible combinations of all input variables, i.e., in our case this design generates all possible configurations of a given software system. This design is suitable only for systems with a small number of features, since it generates 2^{N_f} configurations for N_f binary features.

Fractional factorial design is more suitable for systems with a large number of features N_f. This design selects only a fraction of configurations generated by a full factorial design, thus saving experimentation effort. However, when using this design some information is inevitably lost, which causes *confounding* or inability to capture some higher-order feature interactions. *Resolution* of a fractional factorial design defines the level to which main effects or lower-order interactions are confounded with higher-order interactions, i.e., how well we can assess or model main effects and lower-order interactions. For example, fractional factorial design of resolution VI provides enough information to estimate main effects and two-factor feature interactions unconfounded by four-factor (or less) and three-factor (or less) interactions respectively, what is a very precise assessment.

Another question is how many configurations should we sample to build a precise performance prediction model? Valov et al. (2015) used samples of sizes $T \times N_f$ to evaluate different performance prediction models, where T is a training coefficient which can take values in $\{1, \ldots, 5\}$, and N_f is the number of features available in the configurable software system under test. It was demonstrated (Valov et al., 2015) that measuring $3 \times N_f$ random configurations permitted construction of models with high performance prediction accuracy using regression trees for the majority of studied systems. We use the same heuristic $T \times N_f$, where $T = \{3, 4, 5\}$. We found that this provides sufficient coverage of the feature space for construction of accurate performance prediction models.

We denote a regression tree model by a function RT trained using a small random sample of configurations $\mathbf{C}_S$ of size $T \times N_f$ and their actual performance values $A_{S,m_{trn}}$, measured on a training machine m_{trn}, which predicts the performance value $p_{\mathbf{c},m_{trn}}$ on the machine m_{trn} for a specified configuration $\mathbf{c}$:

$$RT(\mathbf{C}_S, A_{S,m_{trn}}, \mathbf{c}) = p_{\mathbf{c},m_{trn}} \tag{3.1}$$

Next, we must select a metric for assessing the prediction accuracy of the trained prediction model RT, and a validation method to prevent overfitting. We use *mean relative error (MRE)* as a metric for evaluating prediction accuracy. *Relative error (RE)* is the relative difference between an actual performance value $a_{\mathbf{c}}$ and a predicted performance value $p_{\mathbf{c}}$ for a particular configuration $\mathbf{c}$:

$$RE(\mathbf{c}) = \frac{a_{\mathbf{c}} - p_{\mathbf{c}}}{a_{\mathbf{c}}} \times 100\% \tag{3.2}$$

Mean relative error is the average of the relative errors calculated for each individual configuration $\mathbf{c}_i$ of a particular sample of configurations $\mathbf{C}_S$,

$$MRE(\mathbf{C}_S) = \frac{\sum_{i=1}^{N_c} RE(\mathbf{c}_i)}{N_c} \tag{3.3}$$

where N_c is a total number of configurations in the sample $\mathbf{C}_S$.

Finally, we must select a validation method. As mentioned previously, we use random samples of configurations $\mathbf{C}_S$ of a fixed size $T \times N_f$. These configurations are sampled from the set of measured configurations $\mathbf{C}_{exp}$. Since the size of the training sample $\mathbf{C}_S$ is fixed, a natural model validation strategy is *holdout validation*. This method separates all available data, $\mathbf{C}_{exp}$, into a training set, $\mathbf{C}_S$, and a testing set, $\mathbf{C}_{exp} \setminus \mathbf{C}_S$. We train a performance prediction model RT using the training set $\mathbf{C}_S$, and assess prediction accuracy of the model using MRE over the testing set: $MRE(\mathbf{C}_{exp} \setminus \mathbf{C}_S)$.

It is worth mentioning that in an industrial setting a different validation method might be required, i.e., when the cost of performance measuring is extremely high and it is not desirable to measure all $T \times N_f$ configurations upfront or simply to have an extra set of measured configurations available. Practitioners could start with a very small training sample $\mathbf{C}_S$ and progressively train their models until they are satisfied with its accuracy. An effective validation method for these low sample sizes is *leave-one-out cross-validation (LOOCV)*. *LOOCV* separates all available data $\mathbf{C}_S$ into two sets: a testing set, consisting of only one configuration $\mathbf{c}_i$, and a training set, consisting of all other configurations $\mathbf{C}_S \setminus \mathbf{c}_i$. A prediction model RT is then trained using $\mathbf{C}_S \setminus \mathbf{c}_i$ and assessed using relative error over $re_i = RE(\mathbf{c}_i)$. This process is repeated for all possible combinations of training sets and testing sets. The overall accuracy of the prediction model RT for the sample $\mathbf{C}_S$ can be assessed by averaging all individual relative errors: $\sum_{i=1}^{N_c} re_i / N_c$.

3.2.2 Training The Transfer Model

To reuse the previously generated performance prediction model built for m_{trn} for performance prediction on a target machine m_{tgt} a practitioner must train a transfer model. We use linear

regression models as our transfer models since we found that they provide good approximations of the transfer function (see Section 3.3.5 for more details).

The samples we use for linear models training should contain configurations that are measured on both the m_{trn} and m_{tgt} hardware platforms. From the steps described in Section 3.2.1, we have a training sample, $\mathbf{C}_S$, of configurations measured on machine m_{trn} of size $T \times N_f$. Instead of measuring a completely new sample of configurations on both m_{trn} and m_{tgt} machines for training the linear model, we can measure the same configurations from $\mathbf{C}_S$ on the target machine m_{tgt}. In this way, we acquire a training sample $\mathbf{C}_S$ of size $T \times N_f$ measured on both m_{trn} and m_{tgt}.

However, measuring all $T \times N_f$ configurations on the target machine m_{tgt} may be prohibitively expensive. Instead, we measure only a subset of $\mathbf{C}_S$ on both machines $\mathbf{C}_{both} \subset \mathbf{C}_S$. We populate $\mathbf{C}_{both}$ by selecting at least five configurations from $\mathbf{C}_S$ using Sobol sampling (Sobol and Levitan, 1999) (see Section 3.3.6 for more details).

Using the sample $\mathbf{C}_{both}$ we can build a model to transfer performance prediction results from m_{trn} to m_{tgt}. We use a simple linear regression model as a transfer model since it provides good approximation of transfer functions between different machines in our case study (see Section 3.3.5 for more details). This linear model L, given a performance value $p_{\mathbf{c},m_{trn}}$ for a configuration $\mathbf{c}$ on the machine m_{trn}, can predict performance value $p_{\mathbf{c},m_{tgt}}$ of $\mathbf{c}$ on the machine m_{tgt}:

$$L(p_{\mathbf{c},m_{trn}}) = \alpha + \beta \times p_{\mathbf{c},m_{trn}} = p_{\mathbf{c},m_{tgt}} \tag{3.4}$$

3.2.3 Transferring Prediction Results

In the previous steps we selected training and target machines (m_{trn} and m_{tgt}), built a performance prediction model RT based on a small sample $\mathbf{C}_S$ of configurations measured on m_{trn}, and built a linear transfer model L based on a small subsample $\mathbf{C}_{both} \subset \mathbf{C}_S$ of configurations measured on both m_{trn} and m_{tgt} machines. To transfer the prediction model RT to m_{tgt} we just need to transform the predictions of RT using the linear transfer model L. For example, we have a configuration $\mathbf{c}$ that is not measured neither on m_{trn} nor on m_{tgt} machines. To compute $p_{\mathbf{c},m_{tgt}}$ we can use the following equations:

$$p_{\mathbf{c},m_{trn}} = RT(\mathbf{C}_S, A_{S,m_{trn}}, \mathbf{c}) \tag{3.5}$$

Then we can use L to assess performance of $\mathbf{c}$ on m_{tgt}:

$$p_{\mathbf{c},m_{tgt}} = L(p_{\mathbf{c},m_{trn}}) \tag{3.6}$$

3.3 Evaluation

In order to evaluate our approach, we address the following research questions through a set of experiments:

RQ1 How accurate are the transferred performance models created using the process described in Section 3.2? (Section 3.3.2)

RQ2 How does model accuracy vary between different configurable software systems? (Section 3.3.3)

RQ3 How fast is the process of transferring performance prediction models? (Section 3.3.4)

RQ4 Why does the proposed approach work or are the results accidental? (Section 3.3.5)

RQ5 What is an optimal way of building the linear transfer model? (Section 3.3.6)

3.3.1 Experimental Setup

Subject Systems

We measure the performance impact of several different features of 3 different software systems, XZ (Collin, 2020), x264 (VideoLAN Organization, 2020a), and SQLite (Hipp, 2020). These systems represent several common tasks performed by applications: compression of data, transformation of media, and interaction with a database. XZ is a compression utility for UNIX-like operating systems which uses LZMA2 compression. x264 is a library and utility for encoding video streams into the H.264/MPEG-4 AVC compression format. SQLite is a library and application that implements a file-oriented SQL database and is a popular choice for application file formats due to its flexibility.

Each feature that we varied was chosen either based on previous experiments in feature performance regression (Guo et al., 2013), or system documentation and preliminary experiments. For XZ, we measure performance effects from features, such as, varying the "extreme" parameter, varying the "sparse output file" option and applying constraints on memory usage. For x264, we measure performance effects from turning on and off different assembly optimizations, varying the "frame-lookahead", and varying partition search types. For SQLite, we measure performance effects from varying the "synchronous" option, varying the journalling strategy, and varying the amount of space available to mmap.

Subject Hardware Platforms

We carried out our system performance measurements on DataMill (Petkovich et al., 2015), a distributed heterogeneous performance evaluation platform. Each machine of DataMill was setup with identical software and executes Gentoo Linux (Kernel version 3.8.13). Only the DataMill worker software, a kernel, boot manager, and logging daemon were installed on each machine on top of the base set of Gentoo packages, resulting in a minimal set of software. Table 3.2 summarizes hardware configurations of DataMill machines used for our experiments.

Although we did have an exclusive access to DataMill machines, we only had a limited time to use DataMill itself since it is used by many research groups. Therefore we weren't able to measure each configurable system on the whole DataMill cluster, but only on a subset of machines. Table 3.2 shows which machines were used for measuring performance of different software systems.

Table 3.2: Summary of hardware platforms on which configurable software systems were measured; MID – Machine ID in DataMill cluster; NC – Number of CPUs; IS – Instruction set; CCR – CPU clock rate (MHz); RAM – RAM memory size (MB)

Systems			Machines				
XZ	x264	SQLite	MID	NC	IS	CCR	RAM
✓			73	2	i686	1733	1771
✓	✓	✓	75	2	i686	3200	977
✓			77	2	i686	2992	2024
✓			78	1	i686	1495	755
✓			79	4	x86_64	3291	7961
✓			80	8	x86_64	3401	7907
✓	✓		81	16	x86_64	2411	32193
	✓		87	1	i686	1595	249
	✓		88	1	i686	1700	978
		✓	90	2	i686	3200	977
	✓		91	1	i686	2400	1009
	✓	✓	97	2	i686	2992	873
	✓	✓	98	2	i686	2992	873
		✓	99	2	i686	2793	880
	✓		103	2	i686	3200	881
	✓		104	1	i686	1800	502
	✓	✓	105	2	i686	3200	881
	✓		106	2	i686	3192	494
		✓	125	4	x86_64	3301	7960
	✓		128	2	i686	2993	2024
		✓	130	2	i686	3198	880
		✓	146	2	i686	2998	872
		✓	157	36	x86_64	2301	15954

Due to constraints on experiment bandwidth on DataMill, and differing support for features on certain platforms, some feature-configuration/hardware-platform combinations were not measured. Moreover, not all experiment trials terminated correctly; thus only a subset of desired configurations were measured across all machines. Table 3.3 provides a summary of available machines and measured configurations for each configurable system. The scripts used to perform the experiment trials are available on the DataMill website for the purposes of experiment reproduction.

Table 3.3: Summary of measured systems; $\mathbf{N_f}$ – Number of features; NM – Number of machines on which systems were measured; NMC – Number of measured configurations

System	N_f	NM	NMC
XZ	7	7	154
x264	7	11	165
SQLite	5	10	32

Table 3.4: Mean Relative Error (%) of transferred performance models of XZ system, built using different sampling sizes on training and target machines

Target Machines	Sampling Sizes	Training Machines											
		Machine №75			Machine №78			Machine №80			Machine №81		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	5.8	2.0	1.4	9.5	5.4	5.0	8.0	5.2	4.5	6.8	4.1	3.4
75	10	5.2	2.1	1.1	7.2	4.6	3.8	8.1	5.0	4.7	6.9	3.8	3.7
75	15	5.7	2.3	1.7	6.8	4.3	3.3	8.8	5.4	4.4	7.3	4.0	3.1
78	5	8.5	5.7	4.9	6.6	3.0	1.6	9.9	7.3	6.7	8.7	6.9	6.1
78	10	6.6	5.0	3.8	6.5	3.0	1.5	8.1	5.4	4.6	7.7	4.5	3.6
78	15	7.3	3.6	3.6	8.4	3.3	1.8	7.9	5.4	4.6	7.2	4.2	3.9
80	5	10.2	6.9	5.8	11.9	9.6	9.4	9.5	4.8	1.6	10.6	3.8	3.4
80	10	8.2	6.3	5.5	10.6	6.8	5.6	8.2	2.0	1.9	7.1	3.4	2.7
80	15	10.4	6.5	5.1	11.0	6.4	5.6	8.5	4.3	2.3	7.2	3.8	2.6
81	5	9.0	5.9	4.3	11.5	8.2	7.1	9.7	3.6	3.1	7.9	4.1	1.6
81	10	8.9	5.1	4.2	8.7	5.2	4.5	8.8	3.4	2.8	6.5	4.2	1.9
81	15	8.6	5.0	4.0	10.1	5.1	4.5	9.6	3.2	2.6	10.0	2.9	1.5

Measurements and Sampling

For each software system, we analyse the effect of choosing training and target machine pairs and the effect of varying the size of the performance training sample $\mathbf{C}_S$ and the transfer training sample $\mathbf{C}_{both}$ on model accuracy. The sample $\mathbf{C}_S$ varies in $\{3 \times N_f, 4 \times N_f, 5 \times N_f\}$, and the sample $\mathbf{C}_{both}$ varies in $\{5, 10, 15\}$.

As mentioned in Section 3.3.1, we measured each studied configurable system on multiple platforms as shown in Table 3.2 and Table 3.3. However, due to space constraints for each system we present results only for a subset of four different training and target machines. The data we obtained from these machines is summarized in Tables 3.4, 3.6, and 3.8.

We examine each combination of the training machine m_{trn}, target machine m_{tgt}, and sampling sizes for both $|\mathbf{C}_S|$ and $|\mathbf{C}_{both}|$, in a full-factorial experiment design (Antony, 2003; Montgomery, 2008). For each configurable system we have 4 training machines, 4 target machines, 3 sizes of $\mathbf{C}_S$, and 3 sizes of $\mathbf{C}_{both}$, which produces a total of $4 \times 4 \times 3 \times 3 = 144$ different test cases. To acquire the mean prediction relative error for each test case, we follow the model transferring process described in Section 3.2 for 100 different randomly sampled $\mathbf{C}_S$ and $\mathbf{C}_{both}$.

3.3.2 Experiment on Prediction Accuracy

To answer RQ1, we present the results of our transferred performance prediction models across different hardware platforms (Table 3.4, Table 3.6, Table 3.8). As we can see from the results, the majority of training and target machine pairs have strong monotonically decreasing trends in their mean relative error with the increase in training sample size from $3 \times N_f$ to $5 \times N_f$. This follows the intuition that more training data results in more accurate performance prediction models.

Table 3.5: Mean Relative Error (%) of XZ system added by the transferring process

Target Machines	Sampling Sizes	Training Machines											
		Machine №75			Machine №78			Machine №80			Machine №81		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	0.1	0.0	0.0	5.0	4.5	4.3	5.1	4.0	4.1	4.1	3.0	2.9
75	10	0.1	0.0	0.0	4.3	3.5	3.4	4.5	4.2	4.1	3.5	3.0	3.0
75	15	0.1	0.0	0.0	3.8	3.0	2.9	4.7	3.9	3.7	3.8	3.2	2.7
78	5	5.0	4.3	4.1	0.1	0.0	0.0	6.6	6.1	6.2	5.7	5.4	5.3
78	10	4.1	3.3	2.9	0.0	0.0	0.0	4.4	4.2	3.8	3.7	3.2	2.9
78	15	3.7	2.7	2.6	0.1	0.0	0.0	4.5	3.9	3.8	3.7	3.0	2.9
80	5	5.6	4.6	4.7	8.4	8.1	8.0	0.0	0.0	0.0	3.0	2.6	2.3
80	10	5.2	4.8	4.6	7.2	4.6	4.5	0.1	0.0	0.0	2.8	2.2	2.0
80	15	4.9	4.4	4.4	5.6	4.5	4.6	0.1	0.0	0.0	3.9	2.0	1.9
81	5	5.1	3.6	3.6	7.2	7.5	7.0	3.6	2.7	2.6	0.1	0.0	0.0
81	10	5.1	4.3	3.8	4.6	3.8	3.8	3.3	2.3	2.2	0.1	0.0	0.0
81	15	4.6	3.3	3.4	4.4	3.9	3.8	3.1	2.4	2.1	0.1	0.0	0.0

Table 3.6: Mean Relative Error (%) of transferred performance models of SQLite system, built using different sampling sizes on training and target machines

Target Machines	Sampling Sizes	Training Machines											
		Machine №75			Machine №99			Machine №125			Machine №157		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	0.7	0.5	0.4	0.7	0.6	0.6	1.4	1.3	1.3	2.8	2.6	2.6
75	10	0.8	0.5	0.4	0.7	0.6	0.6	1.3	1.2	1.2	2.6	2.6	2.4
75	15	0.7	0.5	0.4	0.7	0.6	0.6	1.3	1.2	1.1	2.6	2.5	2.4
99	5	0.8	0.7	0.6	0.8	0.6	0.6	1.6	1.4	1.3	2.6	2.6	2.6
99	10	0.8	0.6	0.6	0.9	0.6	0.6	1.5	1.3	1.2	2.6	2.4	2.4
99	15	0.7	0.6	0.6	0.8	0.6	0.6	1.4	1.3	1.2	2.6	2.4	2.4
125	5	1.5	1.4	1.3	1.6	1.5	1.4	1.9	1.5	1.1	2.6	2.4	2.3
125	10	1.4	1.3	1.3	1.5	1.4	1.3	1.9	1.5	1.1	2.7	2.3	2.2
125	15	1.4	1.3	1.2	1.5	1.4	1.3	1.8	1.5	1.1	2.4	2.3	2.1
157	5	3.7	3.5	3.5	3.5	3.4	3.3	3.2	2.9	2.7	4.4	4.1	3.6
157	10	3.4	3.2	3.2	3.2	3.1	3.0	3.0	2.7	2.5	4.2	4.0	3.7
157	15	3.3	3.2	3.2	3.1	3.0	2.9	3.0	2.7	2.5	4.3	3.9	3.5

Table 3.7: Mean Relative Error (%) of SQLite system added by the transferring process

Target Machines	Sampling Sizes	Training Machines: Machine №75 Sampling Sizes			Machine №99 Sampling Sizes			Machine №125 Sampling Sizes			Machine №157 Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	0.0	0.0	0.0	0.6	0.5	0.4	1.1	1.1	1.0	2.6	2.5	2.5
75	10	0.0	0.0	0.0	0.5	0.4	0.4	1.2	1.1	1.0	2.4	2.4	2.4
75	15	0.0	0.0	0.0	0.5	0.5	0.4	1.1	1.0	1.0	2.4	2.3	2.3
99	5	0.6	0.6	0.5	0.0	0.0	0.0	1.3	1.2	1.1	2.6	2.5	2.4
99	10	0.6	0.5	0.5	0.1	0.0	0.0	1.1	1.1	1.0	2.4	2.3	2.2
99	15	0.6	0.5	0.5	0.0	0.0	0.0	1.2	1.0	0.9	2.2	2.1	2.1
125	5	1.3	1.1	1.1	1.4	1.4	1.3	0.0	0.0	0.0	2.3	2.1	2.0
125	10	1.2	1.2	1.1	1.3	1.3	1.3	0.0	0.0	0.0	2.1	2.0	1.9
125	15	1.3	1.2	1.1	1.4	1.2	1.2	0.0	0.0	0.0	2.2	2.1	2.0
157	5	2.9	2.7	2.7	2.8	3.0	2.8	2.7	2.7	2.4	0.0	0.0	0.0
157	10	2.8	2.6	2.5	2.8	2.6	2.6	2.4	2.4	2.3	0.0	0.0	0.0
157	15	2.8	2.7	2.5	2.8	2.6	2.5	2.6	2.3	2.2	0.0	0.0	0.0

Table 3.8: Mean ± Standard Deviation [Mean Confidence Interval] of the relative error (%) of transferred performance models of x264 system, built using different sampling sizes on training and target machines

Target Machines	Sampling Sizes	Training Machines: Machine №75 Sampling Sizes			Machine №81 Sampling Sizes			Machine №88 Sampling Sizes			Machine №103 Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	5.3±9.3 [5.2, 5.5]	3.2±7.0 [3.1, 3.4]	2.0±5.1 [1.9, 2.1]	13.5±10.2 [13.3, 13.6]	13.1±9.7 [12.9, 13.2]	13.0±9.3 [12.8, 13.1]	5.4±9.4 [5.3, 5.5]	3.5±6.3 [3.4, 3.6]	2.4±4.6 [2.3, 2.4]	5.2±8.4 [5.1, 5.4]	3.5±6.3 [3.4, 3.6]	2.4±4.1 [2.3, 2.5]
75	10	5.5±9.8 [5.3, 5.6]	3.2±7.1 [3.1, 3.3]	1.7±4.6 [1.7, 1.8]	12.8± 9.3 [12.6, 12.9]	12.3±8.8 [12.2, 12.5]	12.1±8.8 [12.0, 12.3]	5.0±8.5 [4.9, 5.1]	3.2±6.0 [3.1, 3.3]	2.3±4.8 [2.2, 2.4]	4.7±8.2 [4.6, 4.9]	3.1±5.8 [3.0, 3.2]	2.1±4.0 [2.1, 2.2]
75	15	5.4±9.5 [5.2, 5.6]	3.4±7.2 [3.2, 3.5]	2.0±5.2 [1.9, 2.1]	12.5± 9.1 [12.4, 12.7]	12.1±8.5 [12.0, 12.2]	12.0±8.4 [11.8, 12.1]	5.2±8.8 [5.0, 5.3]	3.3±6.4 [3.2, 3.4]	2.3±4.6 [2.3, 2.4]	4.8±8.3 [4.7, 5.0]	3.0±5.7 [3.0, 3.1]	2.1±4.0 [2.0, 2.1]
81	5	14.3±12.9 [14.1, 14.5]	13.2±10.6 [13.0, 13.4]	12.8±9.8 [12.7, 13.0]	6.0±7.5 [5.9, 6.1]	4.7±6.0 [4.6, 4.8]	3.7±5.3 [3.6, 3.8]	13.7±11.7 [13.5, 13.8]	12.9±10.1 [12.7, 13.0]	12.7±10.0 [12.6, 12.9]	13.6±11.2 [13.4, 13.8]	12.7±9.9 [12.6, 12.9]	12.3±9.2 [12.2, 12.5]
81	10	12.3±10.1 [12.2, 12.5]	11.4± 8.8 [11.3, 11.5]	11.0±8.4 [10.9, 11.2]	5.8±7.1 [5.7, 5.9]	4.7±6.2 [4.6, 4.8]	3.9±5.4 [3.8, 4.0]	12.1± 9.6 [11.9, 12.2]	11.4± 8.7 [11.2, 11.5]	11.0± 8.3 [10.9, 11.2]	11.9± 9.8 [11.7, 12.0]	11.2±8.8 [11.0, 11.3]	10.7±8.2 [10.6, 10.9]
81	15	11.7± 9.2 [11.6, 11.9]	11.0± 8.9 [10.9, 11.1]	10.6±8.1 [10.5, 10.7]	6.3±7.7 [6.1, 6.4]	4.7±6.1 [4.6, 4.8]	3.9±5.5 [3.8, 4.0]	12.2±10.1 [12.0, 12.3]	11.1± 8.5 [10.9, 11.2]	10.8± 7.9 [10.6, 10.9]	11.6± 9.1 [11.5, 11.7]	10.8±8.0 [10.6, 10.9]	10.6±8.0 [10.4, 10.7]
88	5	4.7±7.9 [4.6, 4.8]	3.3±5.8 [3.2, 3.4]	2.3±4.2 [2.2, 2.4]	13.7±10.1 [13.6, 13.9]	12.8±9.8 [12.6, 12.9]	12.4±8.9 [12.2, 12.5]	5.3± 9.2 [5.1, 5.4]	3.8±8.6 [3.6, 3.9]	2.2±5.6 [2.1, 2.3]	5.8±8.2 [5.6, 5.9]	3.9±5.8 [3.8, 4.0]	3.0±3.9 [2.9, 3.1]
88	10	5.0±9.0 [4.9, 5.2]	3.0±5.7 [2.9, 3.1]	2.2±4.2 [2.1, 2.2]	12.3± 9.0 [12.2, 12.5]	12.1±9.1 [11.9, 12.2]	11.8±8.6 [11.7, 12.0]	5.8±10.1 [5.6, 6.0]	3.5±7.3 [3.4, 3.7]	2.0±5.2 [1.9, 2.1]	5.2±8.1 [5.0, 5.3]	3.7±5.8 [3.6, 3.7]	2.8±4.2 [2.7, 2.9]
88	15	5.0±8.6 [4.8, 5.1]	3.3±6.3 [3.2, 3.4]	2.2±4.4 [2.1, 2.3]	12.2± 8.9 [12.1, 12.4]	11.7±8.6 [11.6, 11.9]	11.7±8.4 [11.6, 11.8]	5.4± 9.9 [5.3, 5.6]	3.5±7.2 [3.3, 3.6]	2.1±5.3 [2.0, 2.2]	5.1±8.1 [5.0, 5.2]	3.4±5.3 [3.3, 3.5]	2.7±4.0 [2.7, 2.8]
103	5	5.6±9.2 [5.5, 5.7]	3.6±6.2 [3.5, 3.7]	2.6±4.4 [2.5, 2.7]	14.4±11.0 [14.3, 14.6]	13.4±9.8 [13.3, 13.6]	13.2±9.4 [13.1, 13.4]	6.0±8.5 [5.9, 6.2]	4.5±6.8 [4.4, 4.6]	3.1±4.1 [3.1, 3.2]	5.9±10.1 [5.7, 6.1]	3.7±7.4 [3.6, 3.8]	2.1±5.0 [2.1, 2.2]
103	10	5.3±9.1 [5.1, 5.4]	3.3±6.3 [3.2, 3.4]	2.2±4.1 [2.2, 2.3]	13.2± 9.6 [13.1, 13.4]	12.6±9.0 [12.5, 12.8]	12.8±9.2 [12.7, 13.0]	5.9±9.2 [5.8, 6.1]	3.8±5.9 [3.7, 3.9]	3.1±4.5 [3.0, 3.1]	6.2±10.6 [6.0, 6.4]	3.2±6.6 [3.1, 3.3]	2.1±4.9 [2.0, 2.1]
103	15	5.3±8.9 [5.1, 5.4]	3.3±6.0 [3.2, 3.4]	2.3±4.3 [2.3, 2.4]	13.4±11.3 [13.2, 13.5]	12.8±9.2 [12.6, 12.9]	12.4±8.8 [12.3, 12.6]	5.6±9.0 [5.5, 5.8]	4.0±6.5 [3.9, 4.1]	2.9±4.8 [2.9, 3.0]	5.8± 9.6 [5.6, 6.0]	3.2±6.9 [3.1, 3.4]	2.4±5.7 [2.3, 2.5]

We can also observe a sharp decrease in the mean relative error when the sampling size increases from 5 to 15 configurations. This is expected, as again, more training data generally leads to better model accuracy. However, this trend is not always monotonic and in some special cases doesn't hold at all. These observations agree with our analysis of learning curves of linear transformation models performed in Section 3.3.6. From Figure 3.4 and Figure 3.5 we can see that even samples as

Table 3.9: Mean ± Standard Deviation of the time cost (ms) of building performance prediction and transferring models of x264 system, using different sampling sizes on training and target machines

Target Machines	Sampling Sizes	Training Machines											
		Machine №75			Machine №81			Machine №88			Machine №103		
		Sampling Sizes			Sampling Sizes			Sampling Sizes			Sampling Sizes		
		$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$	$3N_f$	$4N_f$	$5N_f$
75	5	5.7±0.9	5.4±0.7	5.9±0.7	5.5±0.5	6.1±0.5	6.3±0.8	5.6±0.5	5.7±0.6	6.1±0.5	6.5±0.5	6.4±1.1	6.7±0.6
75	10	6.0±0.6	6.0±0.6	6.0±0.4	5.3±0.9	5.7±0.5	6.2±0.4	5.5±0.5	5.8±0.4	5.9±0.7	5.8±0.6	6.4±0.8	6.3±0.8
75	15	5.3±0.5	5.5±0.5	6.1±0.7	5.7±0.6	5.8±0.6	6.3±0.6	5.5±0.7	5.8±0.6	5.5±0.5	5.8±0.9	6.5±1.1	6.2±0.6
81	5	5.8±0.9	5.6±0.5	6.0±0.4	5.5±0.5	5.9±0.8	6.0±0.6	5.5±0.7	6.1±0.3	5.9±0.7	5.7±0.6	6.0±1.0	6.2±0.9
81	10	5.5±1.0	5.5±0.5	5.9±0.7	5.3±0.6	5.7±0.5	6.1±0.7	5.9±0.7	6.0±0.8	5.8±0.4	5.9±0.5	7.0±0.8	6.1±0.5
81	15	6.2±0.7	6.1±0.8	6.1±0.7	5.4±0.7	5.8±0.4	6.0±0.9	5.6±0.5	5.9±0.3	5.9±0.5	5.2±0.9	6.3±0.6	6.2±0.7
88	5	6.6±0.5	5.8±0.6	5.8±0.6	5.9±0.5	5.8±0.6	5.8±0.4	5.5±0.8	5.7±0.8	6.4±0.9	5.3±0.5	6.0±0.4	6.2±0.6
88	10	5.9±0.8	6.2±0.7	6.2±0.4	5.4±0.7	5.5±0.5	6.0±0.4	5.5±0.5	6.0±0.6	6.1±0.9	5.1±0.7	6.2±0.9	6.4±0.5
88	15	6.0±0.4	5.9±0.5	6.2±1.0	5.4±0.8	5.9±0.3	6.2±0.6	5.7±1.0	5.8±0.6	5.8±0.6	5.2±0.7	5.9±0.7	6.7±1.4
103	5	5.7±0.6	5.9±0.7	6.0±0.6	5.5±0.5	6.0±0.4	5.5±0.7	5.8±0.4	5.7±0.5	6.0±0.4	5.4±0.5	5.5±0.7	6.3±0.8
103	10	5.7±0.8	5.6±0.7	6.3±0.6	5.7±0.8	6.0±0.0	6.2±1.0	5.5±0.5	5.8±0.6	6.3±0.8	5.2±0.9	5.8±0.7	6.5±0.9
103	15	5.6±0.7	5.9±0.5	6.2±0.6	5.8±0.7	5.7±0.5	6.4±0.9	5.7±0.8	6.0±0.6	6.4±0.9	5.6±0.8	6.0±0.8	6.2±0.7

small as 5 configurations can provide very good approximations of the linear transfer model due to the simplicity of linear models. However, when training with small sample sizes (in our experience, on the interval $[5, 20]$) the linear transfer model may get stuck in a local cost-minimum where the generated model may be biased. This is the reason for the non-monotonically decreasing error we observe in Tables 3.4, 3.6, and 3.8.

Table 3.8 shows not only mean values of the relative errors from our models, but also standard deviations and confidence intervals at the 95% confidence level. From Table 3.8 we can see that although our mean relative error is often small, the standard deviations we measure are relatively large and can exceed the mean in absolute value. However, confidence intervals for the mean value, calculated using bootstrapping (Antony, 2003; Montgomery, 2008), are narrow and are almost always less than or equal to 0.5% of the mean relative error.

The data obtained from our experiments $\mathbf{C}_{exp}$, described in Section 3.2.1, is sufficient to capture feature interactions up to order three. However, we built prediction models RT using only small samples $\mathbf{C}_S \subset \mathbf{C}_{exp}$. Therefore, although sample $\mathbf{C}_S$ may permit making a good approximation of its corresponding performance distribution, it is possible that our performance models, $RT_1, RT_2, \ldots, RT_n$, simply cannot capture all feature interactions that are captured by $\mathbf{C}_{exp}$. This creates a situation where RT produces precise performance predictions for the majority of tested configurations $\mathbf{C}_{exp} \setminus \mathbf{C}_S$ (less that 1% relative error), but for some configurations, which contain uncaptured feature interactions, RT can produce very inaccurate predictions (with more than 50% relative error). This is why we see low mean relative errors, high standard deviations (because of the small set of very large outliers), and very narrow confidence intervals (since it is hard for bootstrapping to capture these outliers).

We can assess accuracy of transferred performance models from a slightly different perspective. We can evaluate how much worse are transferred models compared to "native" models, generated specifically for a target hardware platform. To achieve that, for each performance model trained on m_{trn} and transferred to m_{tgt}, we generate a "native" performance model using the same set of configurations which were measured on m_{tgt}. Then we calculate mean relative error of the native model and subtract it from the mean relative error of the transferred model, thus assessing how much we lose in accuracy when transferring a model from a different platform. Results of this assessment

are presented in Table 3.5 and Table 3.7. One can notice that added mean relative error, when transferring prediction models to the same machine, has a non-zero value. This is caused by the fact that implementation of CART that we use in our study is not exactly deterministic and in some special cases it might generate slightly different prediction models from the same training data. This causes different predictions by these models and thus non-zero difference of mean relative prediction error.

In summary, prediction accuracy generally improves with increasing sampling sizes on training and target machines. However, the proposed approach relies on good sampling strategies for generating the training data $\mathbf{C}_S$ such that it captures important feature interactions. This may cause problems in a practical setting where the training sample $\mathbf{C}_S$ might not follow any feature-coverage criteria. We suggest using experimental design for generating samples of configurations for measurement on target machines as they will maximize the amount of information available in the training sample $\mathbf{C}_S$.

3.3.3 Experiment on System Comparison

To answer RQ2 we compare Tables 3.6 and 3.8. As we can see from these tables the mean relative errors for x264 are much higher than those of SQLite. The same process of performance model transfer produces different prediction accuracy for different configurable systems. This is not unexpected as we can generally expect that different systems will have varying levels of predictability in the performance effects of their features, and in the performance effects of the interactions between features. In the case of x264, many features, such as the size of the window used for a filter, or the number of passes used for encoding, have compounding effects with other features. Configurable features of x264 have many complex interactions that can geometrically increase or decrease its encoding performance. Furthermore, for special cases like video encoders, many chipsets include on-board hardware decoding support, further complicating accurate prediction of feature performance across different hardware platforms. On the other hand, simpler software systems like SQLite have far fewer features and many that do not interact significantly, it is much simpler to predict as a result.

3.3.4 Experiment on Time Cost

Toward answering RQ3, Table 3.9 shows the execution time of building both performance prediction models RT and performance transfer models L for different training sample sizes $|\mathbf{C}_S|$ and $|\mathbf{C}_{both}|$. We can see from this table that the amount of training time necessary for building both prediction and transfer models is a small fraction of the time necessary to benchmark individual configurations, let alone exhaustively exploring all feature setting combinations in $\mathbf{C}$ on even a modest sized benchmark of a given software system. For comparison, Table 3.1 shows examples of the amount of time necessary for benchmarking individual configurations of x264.

3.3.5 Exploratory Analysis

Toward answering RQ4, we conduct a thorough analysis of our methodology. We investigate the performance distributions of configurable systems deployed on multiple hardware platforms, compare the structure of performance models trained on different hardware platforms, and show that linear models are effective approximations for performance transfer models. Therefore we show that the accuracy of our results is not accidental or a result of over-fit, and provide explanations of why our approach works.

Analysis of Performance Distributions

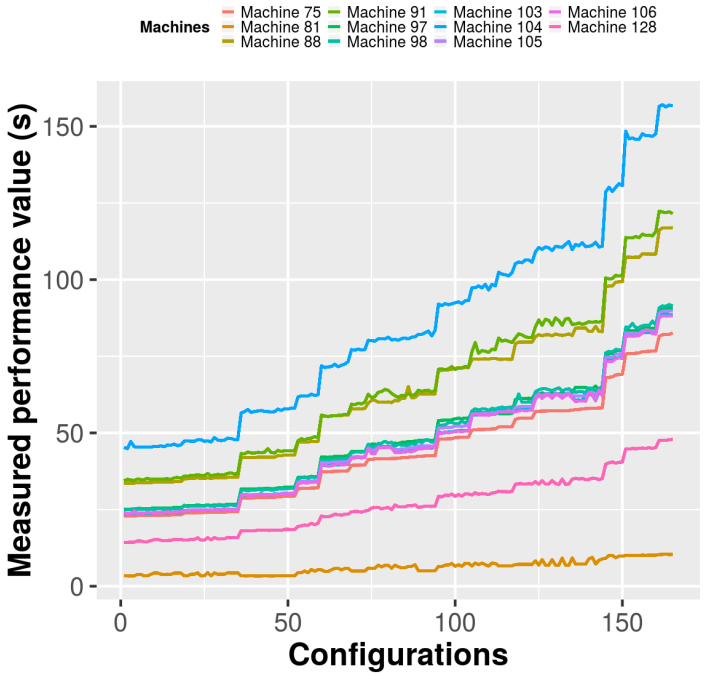


Figure 3.1: Performance distributions of x264 deployed on different machines

To assess the feasibility of transferring performance models of systems between different hardware platforms, we analyzed the similarity of their performance distributions. Studied systems have many features, thus their feature spaces are highly multidimensional and difficult to represent in a manner readily interpretable by the human eye. To visualise the performance distributions of our systems, we take a sample of the configurations that we measure and sort them by the performance of one of the benchmarked hardware platforms.

Figure 3.1 presents performance distributions of x264 deployed on different hardware platforms. We can see that almost all distributions have very similar shapes, although different in absolute values. Though it is only a cursory analysis of the similarity of the performance distributions of our systems across machines, it does give us confidence that even simple polynomial transformations between these distributions could give us good predictions between hardware platforms. There is a

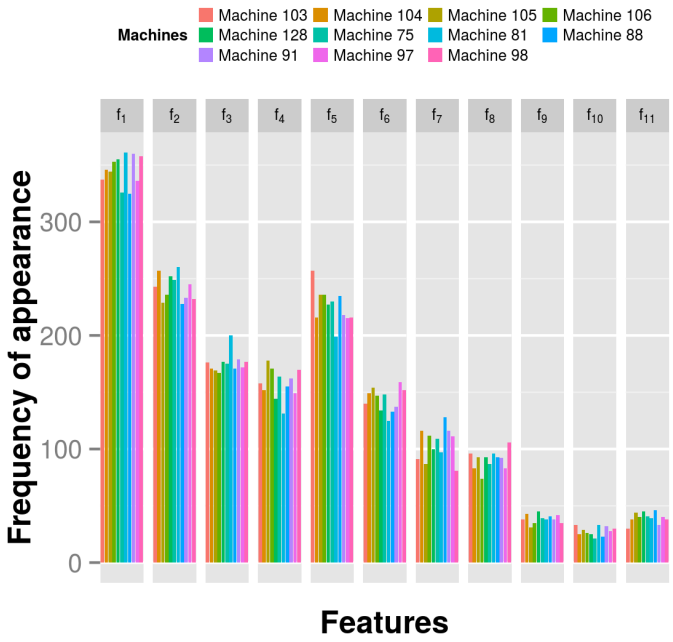


Figure 3.2: Feature distributions of regression trees trained for performance prediction of x264 on different machines

clear pattern indicating that configurations retain their relative performance profile across different hardware platforms, i.e., configuration with low relative performance on one platform will have low relative performance on another platform.

Comparison Analysis of Regression Trees

Regression trees are built by recursively partitioning training dataset into subsets using dataset features. Therefore, features used for dataset partitioning play a major part in defining the structure of a regression tree. Listing all features used in the nodes of a regression tree can be used as a metric for comparing the structure of two different regression trees. Thus by using this metric we can assess the similarity of two regression trees built for the same configurable system, but deployed on different hardware platforms.

Following this logic, we built a feature distribution of regression trees trained for performance prediction of a system deployed on different platforms. From Figure 3.2, we can see that the distributions of features used by trees on different hardware platforms are very similar to each other. From this we can conclude that the structure of the trees themselves are similar across the different hardware platforms we use in our experiments. Therefore it should be possible for us to train a regression tree for performance prediction of a configurable system on one platform and reuse this tree, with small modifications, on another platform.

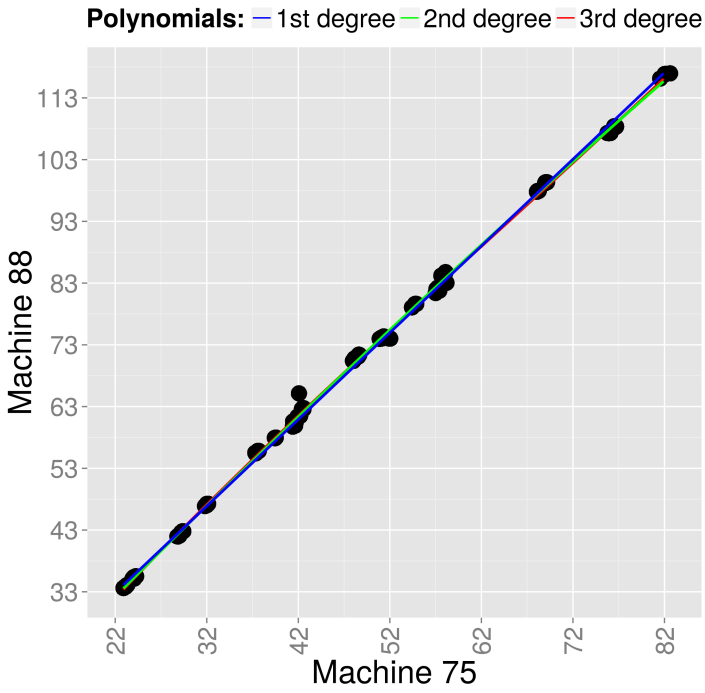


Figure 3.3: Transformation between performance distributions of x264 system deployed on Machine №75 and Machine №88

Analysis of Distributions Transformations

To select a method for transferring prediction models across different platforms we investigated *transfer models* between training and target machine distributions. We used visualizations of the relationships between training and target machine performance distributions to guide the selection of the models used for transfer. An example of the visualizations we used is shown in Figure 3.3. The x-axis corresponds to a configuration's performance on the training platform, while the y-axis corresponds to that same configuration's performance on the target platform.

By exploring several possible transfer models for all systems in our case study, we found that a polynomial regression model provides a good approximations of the transfer function between machines. To evaluate our hypothesis, we fitted three polynomial models to the transformation data: 1^{st}, 2^{nd} and 3^{rd} degree polynomials. For all those software systems and hardware platforms that we tested, we found that a 1^{st} degree polynomial provides an excellent fit of our transformation data, while 2^{nd} and 3^{rd} degree polynomials appear to cause overfitting and unnecessary complications of the transfer model.

Summary

We believe that our proposed process of performance model transfer achieved high prediction accuracy as a result of several main factors. Firstly, the studied configurable systems have very similar performance distributions when deployed on different hardware platforms. Secondly, the transfer

function between these distributions is simple and can be easily approximated using a linear model. Finally, the prediction models trained on the studied software systems have very similar structure when built independently on different platforms. All of these factors together allowed us to use simple and robust methods for performance prediction and model transfer, which resulted in high accuracy achieved by our proposed approach.

3.3.6 Building Linear Transfer Models

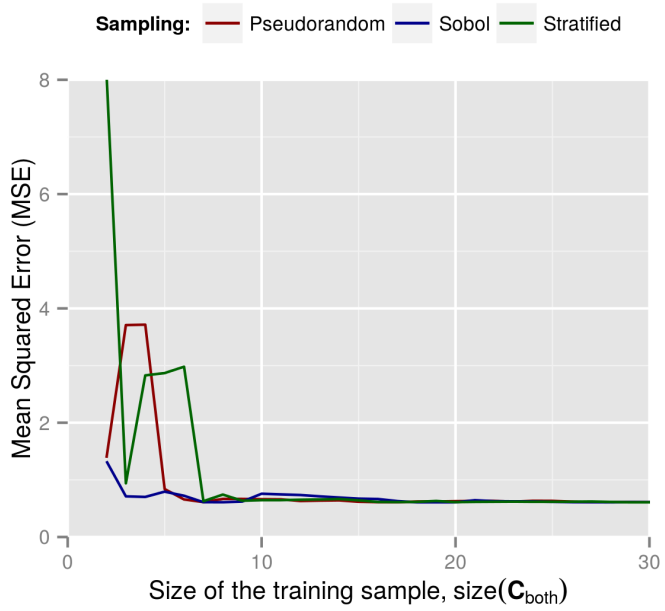


Figure 3.4: Learning curves of a linear transformation between performance distributions of x264 system on Machine №75 and Machine №88

To answer RQ5 we performed a thorough analysis of transfer model building process and tried to answer several important questions. (1) Is it possible to measure only a subset of $\mathbf{C}_S$ on both machines $\mathbf{C}_{both} \subset \mathbf{C}_S$ and build a reliable linear transfer model? (2) Which sampling method to use for $\mathbf{C}_{both}$ to achieve acceptable results faster? (3) Is it possible to figure out a minimum amount of configurations to measure on both m_{trn} and m_{tgt} machines? (4) What is the amount of measured configurations after which additional measurements are not necessary? Toward answering these questions, we decided to analyse the learning curves of the linear transfer models.

We evaluated three different methods of sampling $\mathbf{C}_{both}$: Walker's alias sampling (Ripley, 2009), stratified sampling (Press et al., 1992) and Sobol sampling (Sobol and Levitan, 1999). *Walker's alias sampling* (Ripley, 2009) is a random sampling method which is the default sampling strategy in the R programming language. Walker's alias sampling is an example of a classical pseudo-random sampling method that generates new samples according to a specified probability distribution. *Stratified sampling* (Press et al., 1992) is a random sampling method that exhaustively divides a sampled

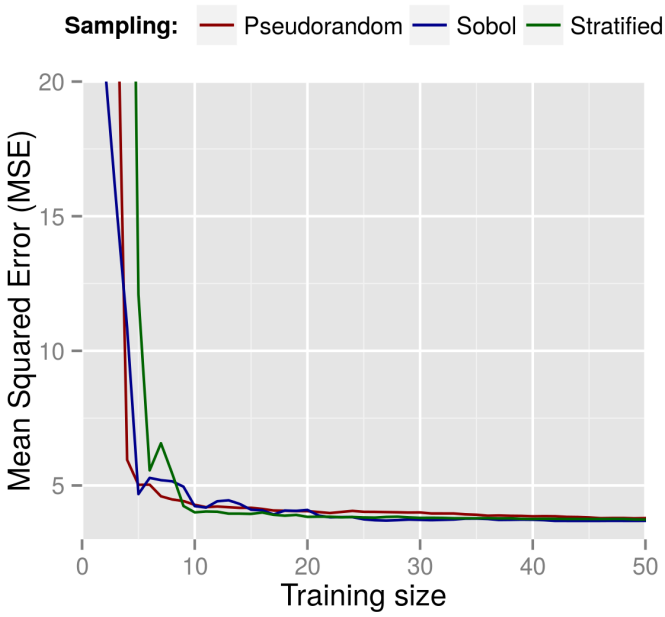


Figure 3.5: Average learning curve of a linear transformation between performance distributions of x264 system

population into mutually exclusive subsets of observations before performing actual sampling. This allows to cover the sampled population more evenly, which in some cases significantly improves representation of the whole population by a sample. *Sobol sampling* (Sobol and Levitan, 1999) is an example of a quasi-random sampling method. Sobol sampling is similar to pseudo-random sampling, as it generates new samples with respect to a given probability distribution, however quasi-random methods are specifically designed to cover a sampled population more uniformly than pseudo random strategies.

To generate a linear transfer model between the performance distributions of machines m_{trn} and m_{tgt}, we build a training dataset using all available configurations $\mathbf{C}_{exp}$. An example of this training data and the resulting linear transfer model is shown in Figure 3.3.

The algorithm we used for building learning curves for linear transfer models is as follows:

1. Initialize $\mathbf{C}_{both}$ by randomly sampling a configuration from $\mathbf{C}_{exp}$.
2. Randomly sample another configuration from the set $\mathbf{C}_{exp} \setminus \mathbf{C}_{both}$ and add it to the training sample $\mathbf{C}_{both}$.
3. Build a linear model L based on the sample $\mathbf{C}_{both}$.
4. Assess how well L approximates the transformation by using *mean squared error (MSE)* over the full set of configurations $\mathbf{C}_{exp}$. [1].

[1] Unfortunately, the proposed methodology represents a classical example of a data leakage, since sets $\mathbf{C}_{both}$ and

5. If the set $\mathbf{C}_{exp} \setminus \mathbf{C}_{both}$ is non-empty go to the Step 2. Otherwise, build the learning curve of L by combining mean squared errors for different sizes of $\mathbf{C}_{both}$.

Figure 3.4 represents the learning curve of a linear model approximating the transfer function between the performance distributions of Machine №75 and Machine №88 when running the x264 software system. Figure 3.5 shows the average learning curve of a linear transformation between performance distributions of individual machines running the x264 software system.

We gained several key insights through analysis of our averaged learning curves for all studied configurable software systems and learning curves for all combinations of training and target machines. Firstly, it is possible to measure only a small sample of configurations $\mathbf{C}_{both} \subset \mathbf{C}_S$ to build a reliable linear transformation between two performance distributions. Secondly, we noticed that for all systems, the biggest improvement in the performance of our linear transfer models occurs when $size(\mathbf{C}_{both}) \in [2, 10]$. However, when $size(\mathbf{C}_{both}) > 20$ practically no performance improvement from additional samples is observed. As a result, we recommend that the size interval for training linear transfer models be set to $size(\mathbf{C}_{both}) \in [10, 20]$.

3.3.7 Threats to Validity

To enhance internal validity, we implemented automated random sampling of configurations $\mathbf{C}_S$ on training machines and $\mathbf{C}_{both}$ on target machines. As was mentioned in Section 3.3.1, $\mathbf{C}_S$ varies in $\{3 \times N_f, 4 \times N_f, 5 \times N_f\}$, and $\mathbf{C}_{both}$ varies in $\{5, 10, 15\}$. For each combination of these sampling sizes, $\mathbf{C}_S$ and $\mathbf{C}_{both}$ were independently and randomly sampled ten times. Thus resulting mean relative errors presented in Tables 3.4, 3.8, 3.6 are averaged over ten independent transferring experiments. This allowed us to avoid bias caused by selecting training data for prediction and transfer models.

An obvious threat to external validity is that the results are derived from experiments on a limited number of software systems and a limited range of hardware. To reduce the threat we benchmarked three configurable systems with different sizes, number of features and covering different application domains. All of the studied systems are used in real-world settings. When benchmarking subject configurable systems we measured each configuration three times. Thus actual performance values in our study are averages over three independent measurements. This allowed us to address possible measurement error in our experiment.

To further enhance external validity, we measured each system on multiple hardware platforms with different number of CPUs, instruction sets, clock rates and memory sizes (see Table 3.2 for more details). We performed transferring experiments for all possible pairs of machines with differing hardware configurations and presented a subset of these experiments in Tables 3.4, 3.8, 3.6, 3.9.

However, we acknowledge that our experiments investigated a very limited set of software systems and hardware platforms and current results might not extrapolate very well to other hardware and software. We suspect that our approach might not work when transferring performance prediction model of a software system that is specifically designed for a particular hardware configuration.

$\mathbf{C}_{exp}$ overlap. We fixed this problem in the subsequent work and we used *leave-one-out cross validation* technique for assessing property transferring models (see Section 4.2.3 for details)

For example, some software systems might use hardware acceleration, like GPUs, for their tasks. If such a system is deployed on a hardware that doesn't have a dedicated GPU, it's performance distribution might appear completely distorted. Thus linear transformation might not provide a good approximation of a transfer model. This hypothesis should be investigated in future work.

3.4 Summary

In the current chapter we proposed an approach for transferring performance prediction models of configurable software systems across different hardware platforms. We performed a rigorous exploratory analysis of the proposed methodology, including: (1) performance distributions comparison, (2) regression models structure comparison, (3) linear transformation analysis, and (4) comparison of different sampling strategies. We observed a high correlation between performance distributions similarity and high prediction accuracy of our method. We showed that similarity of performance distributions is correlated with structure of performance prediction models. We demonstrated that linear model provides a good approximation of transformation between performance distributions of a system deployed in different hardware environments and showed that it is possible to build a reliable linear transfer model using a small sample of measured configurations $\mathbf{C}_{both}$, where $size(\mathbf{C}_{both}) \in [5, 10]$.

We performed a thorough quantitative analysis of our methodology. We showed that our approach achieves high accuracy (less than 10% mean relative error) for the majority of prediction model transfers. Moreover, we observe a decreasing tendency of prediction error with increase of the training data for prediction or linear transfer models. Finally, we demonstrated that the time required for building both performance prediction and linear transfer models is negligible (less than 10 ms) compared to the time budget required for acquiring configuration measurements.

Chapter 4

Transferring Pareto Frontiers

Software systems provide *configuration options* for end users to provide flexibility in meeting their requirements. Apart from having a direct influence on a system's functional behavior, configuration options usually influence non-functional properties, like runtime performance, memory consumption, and overall computational cost. System's configuration options that are available for tuning to end users of the system are called *features* (Guo et al., 2013). A specific choice of feature values determines a system *configuration*. Thus for each system configuration a user can acquire a set of measured properties values. We regard a configuration as a *Pareto optimal* one, if no other configuration improves one or more properties of the Pareto optimal configuration without degrading at least one other property. All Pareto optimal configurations of a system define a *Pareto frontier* of this system. In other words, a Pareto frontier is a set of system configurations, each of which is optimal in its own specific way.

Identifying the Pareto frontier for a configurable system is challenging. First of all, building the exact Pareto frontier of a system's configuration space requires complete knowledge of properties' values for each configuration. This might be infeasible since: (1) a configuration space usually grows exponentially with the number of features, (2) benchmarking time for a single configuration might be high, (3) while a total benchmarking budget available to a user might be relatively low. Secondly, if a user has to deploy a system across heterogeneous hardware, then benchmarking results previously acquired on one hardware might be irrelevant for another hardware and new measurements of a configuration space might be required.

The first problem of incomplete benchmarking information can be solved by approximation of properties' values for a particular hardware environment. This topic has been thoroughly investigated for various use cases (see Guo et al., 2013; Hoste et al., 2006; Hutter et al., 2014, 2015; Thereska et al., 2010a,b; Valov et al., 2015; Westermann et al., 2012; Zhang et al., 2015, for details). For example, some researchers investigate runtime performance prediction of configurable software systems based on small random samples of measured configurations, or in another data-efficient way (see Guo et al., 2013, 2018; Valov et al., 2015, for details).

The second problem of heterogeneous hardware environments can be solved by transferring gained knowledge about system's properties across hardware platforms. This topic has also gained attention in research community (see Brewer, 1994, 1995; Hoste et al., 2006; Jamshidi et al., 2017a,b; Thereska et al., 2010a,b; Valov et al., 2017, for details). For example, researchers employ machine

learning techniques to transfer knowledge about system performance across a heterogeneous computational cluster or for a simulated robotic system (see Jamshidi et al., 2017a,b; Valov et al., 2017, for details).

In this work we propose a novel approach for transferring Pareto frontiers of optimal configurations of highly configurable software systems across heterogeneous hardware environments. The main goal of our research is to develop a pragmatic methodology that could be applied in real-world scenarios. Taking into account all previously described requirements (see Section 1 for details), we propose a practical black-box approach that uses machine learning methods for approximation and transferring of Pareto frontiers of configurations across heterogeneous hardware environments. This approach (1) builds a predictor model for each system property of interest, based on a sample of configurations measured on a source hardware platform, (2) combines predicted properties values of all configurations into an approximated Pareto frontier, (3) builds a transfer model for each system property of interest, based on a sample of configurations measured on both source and destination hardware, and (4) applies the transfer models to the approximated Pareto frontier to transfer it to the destination hardware.

To sum up, in this work we make the following contributions:

- We propose an approach for approximation and transferring of Pareto frontiers of optimal configurations across heterogeneous hardware environments, described previously.
- We comprehensively benchmarked five different configurable software systems across a heterogeneous collection of 34 hardware environments based on Microsoft Azure cloud infrastructure, to acquire the necessary data for our experiments.
- We implement the proposed approach and demonstrate its generality by evaluating it using the benchmarked software systems. We regard approximated and transferred Pareto frontiers as binary classifiers that categorize all configurations into Pareto optimal and non-optimal ones on a specified hardware. Thus we can assess the quality of these frontiers by using classification evaluation measures (e.g. sensitivity, specificity, and Matthew's correlation coefficient) and by analyzing measures' trends with changes in predictors and transfer models. Our empirical results demonstrate that it is possible to achieve high accuracy of transferred Pareto frontiers, according to the classification measures and trends.

Source code and data to reproduce experiments in the current chapter are available online at https://bitbucket.org/valovp/icpe2020.

4.1 Example and Notation

To formalize the problem of Pareto frontier approximation and transferring, we need to introduce necessary definitions and notations. A *configurable software system* is a system that provides *configuration options*, e.g. compression utilities, video codecs, compilers, etc. Configuration options influence *functional properties* (e.g. compression or encoding algorithm, compilation heuristic, etc.) and *non-functional* (e.g. performance, memory consumption, scalability, etc.) of the respective configurable software system. A *feature* is a configuration option that is pertinent to system consumers, e.g. developers, system administrators, expert users, etc. We denote a particular system

feature by a binary variable $f_i \in \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$, and all system's features by a set of variables $\mathbb{F} = \{f_1, f_2, \dots f_{N_f}\}$, where $N_f \in \mathbb{N}$ is a total number of features of the system. *Configuration* is a unique set of actually assigned values to all N_f features. We denote a configuration by $\mathbf{c}_i \in \mathbb{B}^{N_f}$, and all configurations by a set $\mathbb{C} = \{\mathbf{c}_1, \mathbf{c}_2, \dots \mathbf{c}_{N_\mathbf{c}}\}$, where $N_\mathbf{c} \in \mathbb{N}$ is a total number of valid configurations. Each system has a set of functional or non-functional properties that we denote by $\mathbb{P} = \{p_1, p_2, \dots p_{N_p}\}$, where $N_p \in \mathbb{N}$ is a total number of such properties.

We perform our study on various *hardware environments* that we denote by $h_k \in \mathbb{N}$ which together form a heterogeneous *hardware cluster* $\mathbb{H} = \{h_1, h_2, \dots, h_{N_h}\}$, where $N_h \in \mathbb{N}$ is a total number of hardware environments. Each property p_j from the set $\mathbb{P}$ is expected to vary when measured for the same configuration $\mathbf{c}_i$ across the cluster $\mathbb{H}$. Thus each configuration $\mathbf{c}_i$ has an actual measured property value $y_{\mathbf{c}_i,p_j,h_k}$ for each property p_j on each hardware h_k. We view properties as functions that map hardware environments and configurations to actual measured values:

$$\begin{aligned} p_j &: \mathbb{B}^{N_f} \times \mathbb{H} \to \mathbb{R} \\ p_j(\mathbf{c}_i, h_k) &= y_{\mathbf{c}_i,p_j,h_k} \end{aligned} \tag{4.1}$$

All actual properties' values of a configuration $\mathbf{c}_i$ on a hardware h_k form a vector that we denote by $\mathbf{y}_{\mathbf{c}_i,\mathbb{P},h_k}$:

$$\mathbf{y}_{\mathbf{c}_i,\mathbb{P},h_k} = [\, y_{\mathbf{c}_i,p_1,h_k}, y_{\mathbf{c}_i,p_2,h_k}, \dots, y_{\mathbf{c}_i,p_{N_p},h_k}] \tag{4.2}$$

Actual properties' values of all configurations $\mathbf{c}_i$ of a sample $\mathbb{C}_S \subset \mathbb{C}$ and of the whole population $\mathbb{C}$ on a hardware h_k form the corresponding sets $\mathbb{Y}_{\mathbb{C}_S,\mathbb{P},h_k}$ and $\mathbb{Y}_{\mathbb{C},\mathbb{P},h_k}$:

$$\mathbb{Y}_{\mathbb{C}_S,\mathbb{P},h_k} = \bigcup_{\mathbf{c}_i \in \mathbb{C}_S} \{\mathbf{y}_{\mathbf{c}_i,\mathbb{P},h_k}\} \tag{4.3}$$

$$\mathbb{Y}_{\mathbb{C},\mathbb{P},h_k} = \bigcup_{\mathbf{c}_i \in \mathbb{C}} \{\mathbf{y}_{\mathbf{c}_i,\mathbb{P},h_k}\} \tag{4.4}$$

Since our primary goal is to transfer a Pareto frontier across hardware environments, we need to be able to distinguish between them. We call an environment a *source hardware environment* if we use it to measure actual property values of configurations, to train predictors for each system property, and to approximate a Pareto frontier. We call an environment a *destination hardware environment* if we use it to train transferrers for each system property, and if we transfer the approximated frontier to this environment. We denote source and destination environments by h_{src} and h_{dst} respectively.

Before we can define what Pareto frontier is, we need to introduce notions of *preference* and *domination* between different configurations. We assume that each property p_j has a *preferred direction of values* that can be inferred from a system domain, e.g. a compression software has a property 'compression rate' and higher values of this property are preferred to lower ones. For the sake of the example, let's pretend that for all properties in $\mathbb{P}$ higher values correspond to more preferred values. Then properties in $\mathbb{P}$ are in fact *utility functions* that describe a level of preference of a particular configuration. Thus we can denote that a property value $y_{\mathbf{c},p_j,h_k}$ *is preferred to* or *improves* a property value $y_{\mathbf{c}',p_j,h_k}$ simply by: $y_{\mathbf{c},p_j,h_k} > y_{\mathbf{c}',p_j,h_k}$. We say that a configuration $\mathbf{c}$

dominates a configuration $\mathbf{c}'$ on a hardware h_k, or $\mathbf{c} \succ^{h_k} \mathbf{c}'$, if $y_{\mathbf{c},p_j,h_k} > y_{\mathbf{c}',p_j,h_k}$ for some $p_j \in \mathbb{P}$ and $y_{\mathbf{c},p_j,h_k} \geq y_{\mathbf{c}',p_j,h_k}$ for all $p_j \in \mathbb{P}$.

Finally, we can introduce notions of *Pareto optimality* and *Pareto frontier*. We call a configuration $\mathbf{c}$ a *Pareto optimal* configuration on a hardware h_k, if there is no other configuration $\mathbf{c}'$ that improves one or more properties values of $\mathbf{c}$ without worsening at least one other property value, i.e. $\mathbf{c}$ is not dominated by any other configuration $\mathbf{c}'$. More formally, $\mathbf{c}$ is a *Pareto optimal* configuration on a hardware h_k, if there is no other $\mathbf{c}'$ such that $y_{\mathbf{c}',p_j,h_k} > y_{\mathbf{c},p_j,h_k}$ for some $p_j \in \mathbb{P}$ and $y_{\mathbf{c}',p_j,h_k} \geq y_{\mathbf{c},p_j,h_k}$ for all $p_j \in \mathbb{P}$. The *Pareto frontier* of a system on a given hardware h_k is a set of all Pareto optimal configurations on this hardware that we denote by $\mathbb{C}^{PF}_{h_k}$. In other words, the Pareto frontier is a set of configurations that are not strictly dominated by any other configuration, or more formally:

$$\mathbb{C}^{PF}_{h_k} = \{\mathbf{c} \in \mathbb{C} : \{\mathbf{c}' \in \mathbb{C} : \mathbf{c}' \succ^{h_k} \mathbf{c}, \mathbf{c}' \neq \mathbf{c}\} = \emptyset\} \tag{4.5}$$

Figure 4.1 and Figure 4.2 demonstrate example Pareto frontiers of studied configurable systems. We utilize these systems for the evaluation of our approach for approximation and transferring of Pareto frontiers later in this chapter. Each Pareto frontier on these figures shows trade-offs between *compression time* and *compressed size* system properties. Figure 4.1 demonstrates that Pareto frontiers of studied software systems significantly differ in their structure because of completely different configuration spaces. Figure 4.2 demonstrates that even Pareto frontiers of the same software system can vary significantly across heterogeneous hardware environments.

We have introduced all necessary definitions and notations, thus we can summarize the problem statement in proper terms. We deploy a configurable software system (e.g. XZ) on a hardware platform h_{src}. We select a random training sample of configurations $\mathbb{C}_{trn} \subset \mathbb{C}$ and for each configuration $\mathbf{c}_i \in \mathbb{C}_{trn}$ we acquire an actual value $y_{\mathbf{c}_i,p_j,h_{src}}$ of each property $p_j \in \mathbb{P}$ thus forming a set of measured values $\mathbb{Y}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$. Sets $\mathbb{C}_{trn}$ and $\mathbb{Y}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$ together form a sample of measured configurations $\mathbb{S}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$ on a hardware platform h_{src}. Our goal is to build an approximated Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ based on the sample $\mathbb{S}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$ and to transfer this frontier to all other hardware $\mathbb{H} \setminus \{h_{src}\}$. Thus we can use the Pareto frontier as a binary classifier that separates all configurations into optimal and non-optimal ones, and based on a small random sample of measured configurations $\mathbb{S}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$ classify all configurations $\mathbb{C}$ on all hardware platforms $\mathbb{H}$.

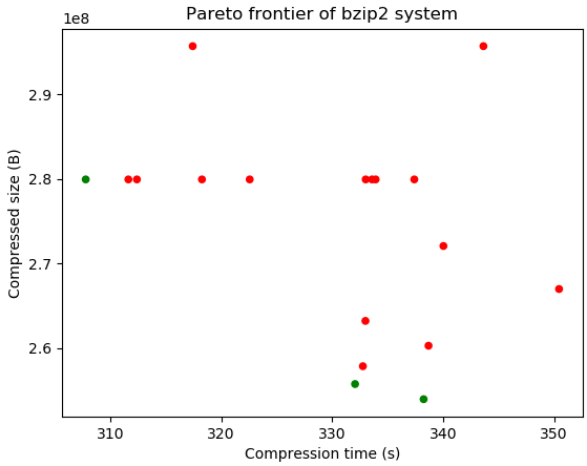
4.2 Transferring Process

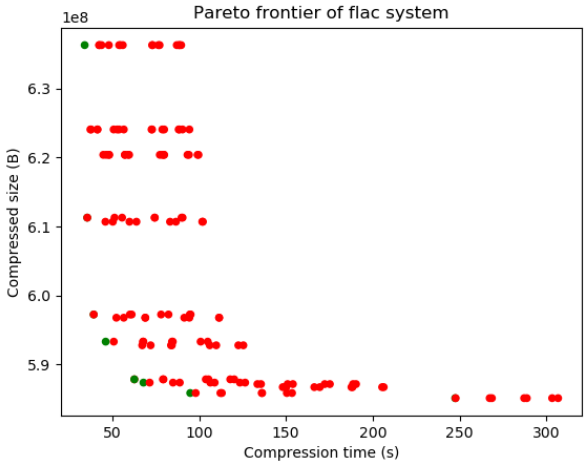
The process of transferring Pareto frontiers can be divided into several main steps: (1) training a predictor for each studied system property (2) building an approximated Pareto frontier using trained predictors (3) training a transferrer for each studied system property (4) transferring the approximated Pareto frontier across hardware platforms using transferrers.

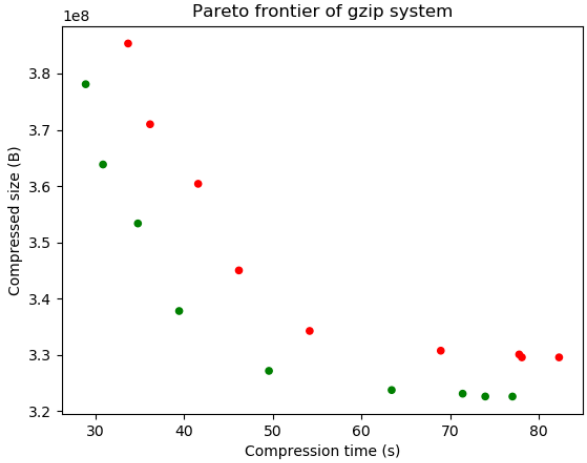
4.2.1 Training Property Prediction Models

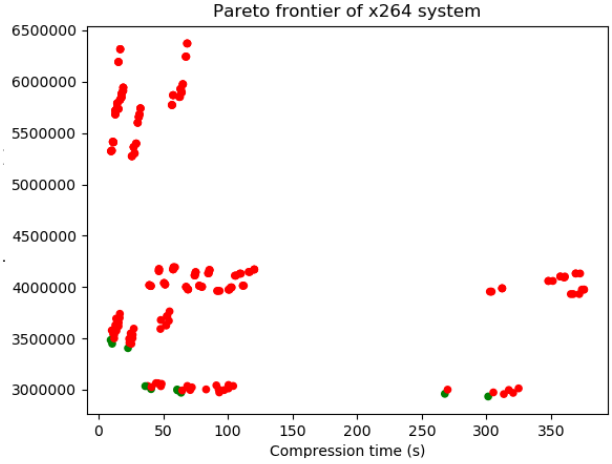
The process of training a property prediction model can be separated into several steps: (1) selecting a data sampling method (2) sampling training data (3) selecting a property predictor model (4)

Figure 4.1: Example Pareto frontiers of studied configurable software systems. Each Pareto frontier shows trade-offs between *compression time* and *compressed size* system properties. Green and red dots denote Pareto optimal and non-optimal configurations respectively.









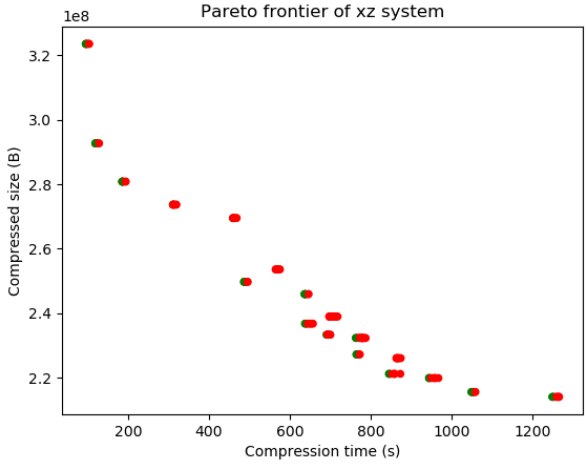
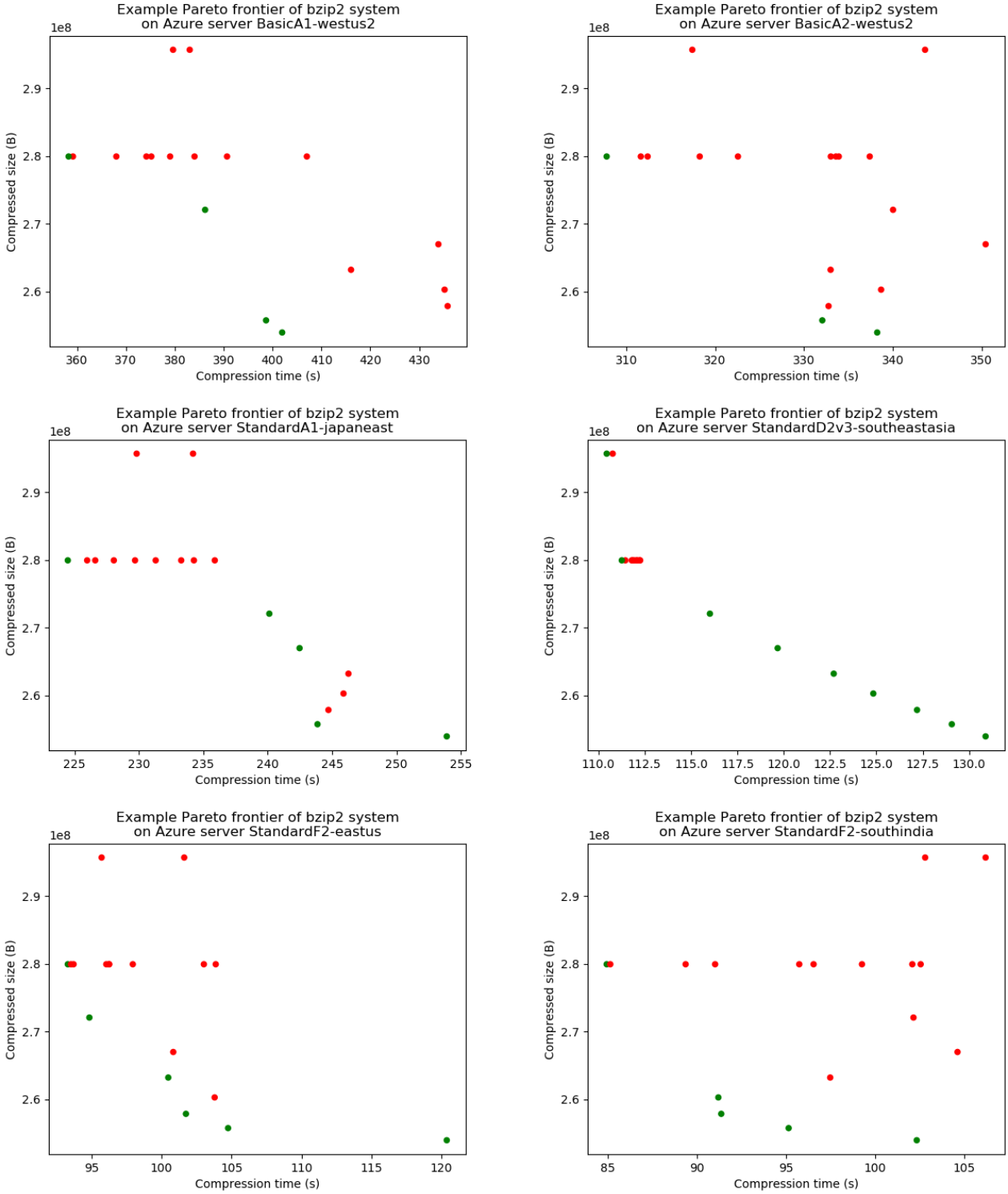


Figure 4.2: Example Pareto frontiers of *bzip2* software system across a heterogeneous cluster of Azure cloud computing environments. Each Pareto frontier shows trade-offs between *compression time* and *compressed size* system properties. Green and red dots denote Pareto optimal and non-optimal configurations respectively.



training a property predictor (5) selecting an evaluation metric and a validation strategy for the trained predictor.

First of all, we need to select a method for sampling training data that will be used for training properties' predictors. There are many different ways in which one can generate a sample of training configurations $\mathbb{C}_{trn}$ from a system's configuration space: pseudo-random sampling, quasi-random sampling, experimental design techniques, and custom sampling heuristics. In the current work we use pseudo-random sampling of configurations in order to mimic the worst-case scenario, when a practitioner does not have any control over selection of training data and available measured configurations might appear completely random. The same paradigm was used in the previous work on performance prediction of configurable software systems by Guo et al. (2013) and Valov et al. (2015). However, in a practical scenario where a practitioner has control over a sampling process, we would advise to use more sophisticated sampling methods like quasi-random sampling and experimental design techniques, since they provide a much more even coverage of the configuration space and should improve quality of a trained predictor. We leave a comparison of different sampling methods for property prediction models' training data for a future work.

Secondly, we need to perform actual data sampling. In the previous work Guo et al. (2013); Valov et al. (2015) researchers used sampling sizes that are multiples of the number of features N_f of the respective system. During evaluation of our approach we've tried all possible sampling sizes in range $[\,2, N_{\mathbb{C}} - 1]$ in order to provide smoother trends for presenting regression and classification measures (see Section 4.3 for details).

Thirdly, we need to choose which model to use as property predictors. During preliminary evaluation of our approach we tried two different regression models: regression trees and random forest. We selected these models as candidates for our solution since they have already showed good results for prediction of configurable systems properties (see Guo et al., 2013; Thereska et al., 2010a,b; Valov et al., 2015; Westermann et al., 2012, for details). Our preliminary experiments demonstrated that unpruned regression tree models provided better prediction results than random forest models. We came to a conclusion that this happens because we work with relatively small training sample sizes. Although random forest model also generates unpruned regression trees to average upon, it trains them using observations resampled with replacement from an original sample provided to the random forest model itself. This approach helps to avoid overfitting when training samples are relatively large, but when training samples are tiny, each individual tree ends up loosing a lot of information. Therefore, in our case a single unpruned regression tree (but trained on a full training sample) will outperform an ensemble of trees that were trained on resampled data.

Fourthly, we have to train property predictors. The only thing left to do, is to select a parameter tuning strategy. We work with unpruned regression trees, i.e. we 'grow' our tree models to a maximal possible size, when each tree has only one observation in each leaf and all available features are used in construction of the tree. We can regard the regression tree predictor as a function RT, that is generated by a *fitting function* $fitRT$. A fitting function, given a small random training sample of configurations $\mathbb{C}_{trn}$ and their measured values $\mathbf{Y}_{\mathbb{C}_{trn},p_j,h_k}$ of a property p_j on a hardware h_k, produces corresponding predictors for the property p_j on the hardware h_k:

$$fitRT(\mathbb{C}_{trn}, \mathbf{Y}_{\mathbb{C}_{trn},p_j,h_k}, \mathbf{c}_i) = RT_{\mathbb{C}_{trn},p_j,h_k} \tag{4.6}$$

Predictor $RT_{\mathbb{C}_{trn},p_j,h_k}$, given a sample of configurations $\mathbb{C}_S$, can predict their values for the property

p_j on the hardware h_k:

$$RT_{\mathbb{C}_{trn},p_j,h_k}(\mathbb{C}_S) = \widehat{\mathbf{Y}}_{\mathbb{C}_S,p_j,h_k} \tag{4.7}$$

Finally, we need to select an evaluation metric and a validation strategy for trained predictors. We have selected *mean absolute percentage error (MAPE)* as a metric for evaluation of RT models since it provides a simple and intuitive measure that is robust to outliers. $MAPE$ can be thought of as a mean of multiple *absolute percentage errors (APE)*. If APE can be expressed as a function that consumes actual $y_{\mathbf{c}_i,p_j,h_k}$ and predicted $\hat{y}_{\mathbf{c}_i,p_j,h_k}$ property values:

$$APE(y_{\mathbf{c}_i,p_j,h_k}, \hat{y}_{\mathbf{c}_i,p_j,h_k}) = \frac{|y_{\mathbf{c}_i,p_j,h_k} - \hat{y}_{\mathbf{c}_i,p_j,h_k}|}{y_{\mathbf{c}_i,p_j,h_k}} \times 100\% \tag{4.8}$$

then $MAPE$ can be expressed as a function that consumes sets of actual $\mathbf{Y}_{\mathbb{C}_S,p_j,h_k}$ and predicted $\widehat{\mathbf{Y}}_{\mathbb{C}_S,p_j,h_k}$ property values:

$$MAPE(\mathbf{Y}_{\mathbb{C}_S,p_j,h_k}, \widehat{\mathbf{Y}}_{\mathbb{C}_S,p_j,h_k}) = \frac{\sum_{i=1}^{N_{\mathbb{C}_S}} APE(y_{\mathbf{c}_i,p_j,h_k}, \hat{y}_{\mathbf{c}_i,p_j,h_k})}{N_{\mathbb{C}_S}} \tag{4.9}$$

where $N_{\mathbb{C}_S}$ is a number of configurations in the set $\mathbb{C}_S$.

We have selected *leave-one-out cross-validation (LOOCV)* as a validation strategy for predictors, since it is especially suitable in a scenario, when the cost of measuring properties for a single configuration is very high and a practitioner wants to minimize the measurement effort. Imagine that a practitioner acquired a sample of configurations $\mathbb{C}_S$ of size $N_{\mathbb{C}_S}$ along with their measured values $\mathbf{Y}_{\mathbb{C}_S,p_j,h_k}$ of a property p_j on a hardware h_k. $LOOCV$ is going to generate out of $\mathbb{C}_S$ sample: $N_{\mathbb{C}_S}$ testing samples $\mathbb{C}^i_{tst} = \{\mathbf{c}_i\}$ that consist of a single configuration and $N_{\mathbb{C}_S}$ training samples $\mathbb{C}^i_{trn} = \mathbb{C}_S \setminus \{\mathbf{c}_i\}$ that consist of all other configurations. Thus $LOOCV$ generates $N_{\mathbb{C}_S}$ pairs of training and testing samples $\{(\mathbb{C}^1_{trn}, \mathbb{C}^1_{tst}), \ldots, (\mathbb{C}^{N_{\mathbb{C}_S}}_{trn}, \mathbb{C}^{N_{\mathbb{C}_S}}_{tst})\}$. Then by training a predictor, e.g. a regression tree RT, on each training sample $\mathbb{C}^i_{trn}$ and by testing the predictor on the corresponding testing sample $\mathbb{C}^i_{tst}$, we can acquire a column vector of predicted property values:

$$\widehat{\mathbf{Y}}_{\mathbb{C}_S,p_j,h_k} = \begin{bmatrix} RT_{\mathbb{C}^1_{trn},p_j,h_k}(\mathbb{C}^1_{tst}) \\ RT_{\mathbb{C}^2_{trn},p_j,h_k}(\mathbb{C}^2_{tst}) \\ \vdots \\ RT_{\mathbb{C}^{N_{\mathbb{C}_S}}_{trn},p_j,h_k}(\mathbb{C}^{N_{\mathbb{C}_S}}_{tst}) \end{bmatrix} \tag{4.10}$$

We can use $\mathbf{Y}_{\mathbb{C}_S,p_j,h_k}$ and $\widehat{\mathbf{Y}}_{\mathbb{C}_S,p_j,h_k}$ to calculate $MAPE$ in order to assess the predictor quality.

4.2.2 Building an Approximated Frontier

The process of building an approximated Pareto frontier consists of the following main steps: (1) approximating all system properties $\mathbb{P}$ for all configurations $\mathbb{C}$ using trained predictors from Section 4.2.1, (2) calculating a Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ based on the approximated configurations' properties.

First of all, we need to acquire all properties' values $\mathbb{P}$ for all configurations $\mathbb{C}$, i.e. $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}$, to assess which configurations are in fact Pareto optimal on a hardware h_{src}, and thus to generate a Pareto frontier. However, we know properties' values only for a small training sample of configurations $\mathbb{C}_{trn}$, i.e. $\mathbf{Y}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$. Therefore, we need to approximate the properties' values for the remaining unmeasured configurations $\mathbb{C}_{rem} = \mathbb{C} \setminus \mathbb{C}_{trn}$, i.e. acquire $\widehat{\mathbf{Y}}_{\mathbb{C}_{rem},\mathbb{P},h_{src}}$. We can obtain $\widehat{\mathbf{Y}}_{\mathbb{C}_{rem},\mathbb{P},h_{src}}$ by systematically predicting all properties in $\mathbb{P}$ for $\mathbb{C}_{rem}$ using corresponding predictors and combining the resulting column vectors into a matrix:

$$\widehat{\mathbf{Y}}_{\mathbb{C}_{rem},\mathbb{P},h_{src}} = \left[RT_{\mathbb{C}_{trn},p_1,h_{src}}(\mathbb{C}_{rem}), \cdots, RT_{\mathbb{C}_{trn},p_{N_p},h_{src}}(\mathbb{C}_{rem})\right] \tag{4.11}$$

We can obtain $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}$ by combining matrices of measured and approximated properties' values:

$$\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}} = \begin{bmatrix} \mathbf{Y}_{\mathbb{C}_{trn},\mathbb{P},h_{src}} \\ \widehat{\mathbf{Y}}_{\mathbb{C}_{rem},\mathbb{P},h_{src}} \end{bmatrix} \tag{4.12}$$

Finally, based on the resulting matrix of all properties' values $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}$ we can build the approximated Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ using any of the known algorithms for exact frontier construction:

$$\widehat{\mathbb{C}}^{PF}_{h_{src}} = PF(\mathbb{C}, \widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}) \tag{4.13}$$

4.2.3 Training Property Transferring Models

The process of training a property transferring model can be separated into the following steps: (1) sampling training data, (2) selecting a property transferrer model, (3) training a property transferrer, (4) selecting an evaluation metric and validation strategy for the trained transferrer.

First of all, to train a property transferrer we need a training sample of configurations $\mathbb{C}_{both}$ that are measured on both source h_{src} and destination h_{dst} hardware. In Section 4.2.1 we've already defined a procedure for acquiring a training sample $\mathbb{C}_{trn}$ measured on h_{src}. Naturally, we can measure configuration from $\mathbb{C}_{trn}$ on h_{dst} as well, thus forming the necessary sample $\mathbb{C}_{both} \subseteq \mathbb{C}_{trn}$.

Secondly, we have to select a model to be used as a property transferrer. During our preliminary evaluation, we have tested multiple machine learning models as property transferrers, and two of them provided the best results overall: simple linear regression model (*SLR*) and unpruned regression tree model (*RT*), discussed previously. We used *SLR*, since it has been already studied in the previous work (see Jamshidi et al., 2017b; Valov et al., 2017, for details) and has demonstrated good results. However, during preliminary evaluation we noticed that *SLR* underperforms for some studied systems, and especially multithreaded ones. On the contrary, *RT* demonstrated better performance overall and significantly better performance for parallel systems. Therefore, unlike previous work (Jamshidi et al., 2017b; Valov et al., 2017) we recommend using *RT* as transferrers, especially when working with multithreaded software.

Thirdly, we have to train the selected transferring models using the measured training sample $\mathbb{C}_{both}$. In the *SLR* model, a line is fit to the training data using classical *ordinary least squares*

(OLS) methodology. OLS performs fitting by minimizing the sum of squared differences between configurations in $\mathbb{C}_{both}$ and the linear model. Thus, for each system property p_j we can acquire a corresponding linear transferrer that given property values on a source hardware will produce corresponding property values on a destination hardware:

$$\begin{aligned} SLR_{p_j}(\widehat{\mathbf{Y}}_{\mathbb{C},p_j,h_{src}}) &= \alpha + \beta \times \widehat{\mathbf{Y}}_{\mathbb{C},p_j,h_{src}} \\ &= \widehat{\mathbf{Y}}_{\mathbb{C},p_j,h_{dst}} \end{aligned} \tag{4.14}$$

The *SLR* training process is completely automatic and doesn't require any parameter tuning. The training process for unpruned regression trees was discussed previously (see Section 4.2.1).

Finally, we need to select an evaluation metric and a validation strategy for the trained property transferring models. Since transferrers might be built using even smaller samples than property prediction models, the most practical validation strategy would still be leave-one-out cross-validation (*LOOCV*). As for an evaluation metric, we again recommend using mean absolute relative error (*MAPE*).

4.2.4 Transferring a Pareto Frontier

During the previous steps of the process (see Section 4.2.1 – Section 4.2.3) we have: (1) selected a source h_{src} and a destination h_{dst} hardware environments, (2) sampled training configurations $\mathbb{C}_{trn}$ and measured all their properties' values $\mathbf{Y}_{\mathbb{C}_{trn},\mathbb{P},h_{src}}$ on h_{src}, (3) trained predictors RT_{p_j} for each property $p_j \in \mathbb{P}$ on h_{src}, (4) predicted all properties' values on h_{src} and acquired $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}$, (5) built an approximated Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ based on $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{src}}$, (6) sampled configurations $\mathbb{C}_{both}$ and measured all their properties' values on both h_{src} and h_{dst}, (7) trained transferrers for each property $p_j \in \mathbb{P}$ to transfer properties' values from h_{src} to h_{dst}.

To transfer the Pareto frontier from h_{src} to h_{dst} we need to perform two steps: (1) approximate all properties $\mathbb{P}$ for all configurations $\mathbb{C}$ on h_{dst}, i.e. acquire $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{dst}}$, and (2) calculate the transferred Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{dst}}$ on h_{dst} based on $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{dst}}$.

First of all, we obtain $\widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{dst}}$ by systematically transferring column vectors of properties' values from h_{src} to h_{dst}, using previously trained property transferrers:

$$\begin{gathered} \widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{dst}} = \\ \left[RT_{p_1}(\widehat{\mathbf{Y}}_{\mathbb{C},p_1,h_{dst}}), \cdots, RT_{p_{N_p}}(\widehat{\mathbf{Y}}_{\mathbb{C},p_{N_p},h_{dst}}) \right] \end{gathered} \tag{4.15}$$

Then, we can calculate the approximated Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{dst}}$ using any known algorithm for exact Pareto frontier construction:

$$\widehat{\mathbb{C}}^{PF}_{h_{dst}} = PF(\mathbb{C}, \widehat{\mathbf{Y}}_{\mathbb{C},\mathbb{P},h_{dst}}) \tag{4.16}$$

4.3 Process Evaluation

To comprehensively evaluate the proposed process of Pareto frontier approximation and transferring, we have formulated the following research questions:

Table 4.1: Summary of all Azure-based virtual machines; CRS – number of CPU cores that are specifically allocated to the virtual machine, RAM – amount of RAM allocated to the VM (GiB), RPC – amount of RAM allocated per CPU core of the VM (GiB), STR – amount of storage allocated to the VM (GiB), STP – storage type allocated to the VM

General Server Info			Server Architecture						Benchmarked Systems				
Name	**Azure Type**	**Deployment Region**	**CPU Model Name**	**CRS**	**RAM**	**RPC**	**STR**	**STP**	**BZIP2**	**GZIP**	**XZ**	**FLAC**	**X264**
BscA0-4171HE	BasicA0	South Brazil	AMD Opteron 4171 HE	1	0.75	0.75	20	HDD		✓			✓
BscA0-2660	BasicA0	East Japan	Intel Xeon E5-2660	1	0.75	0.75	20	HDD	✓	✓		✓	✓
BscA0-2673v3	BasicA0	West US	Intel Xeon E5-2673 v3	1	0.75	0.75	20	HDD	✓	✓		✓	✓
BscA1-4171HE	BasicA1	South Brazil	AMD Opteron 4171 HE	1	1.75	1.75	40	HDD	✓	✓		✓	✓
BscA1-2660	BasicA1	East Japan	Intel Xeon E5-2660	1	1.75	1.75	40	HDD	✓	✓	✓	✓	✓
BscA1-2673v3	BasicA1	West US	Intel Xeon E5-2673 v3	1	1.75	1.75	40	HDD	✓	✓	✓	✓	✓
BscA2-4171HE	BasicA2	South Brazil	AMD Opteron 4171 HE	2	3.5	1.75	60	HDD	✓	✓		✓	✓
BscA2-2660	BasicA2	East Japan	Intel Xeon E5-2660	2	3.5	1.75	60	HDD	✓	✓	✓	✓	✓
BscA2-2673v3	BasicA2	West US	Intel Xeon E5-2673 v3	2	3.5	1.75	60	HDD	✓	✓	✓	✓	✓
StdA0-2673v3	StandardA0	Central Canada	Intel Xeon E5-2673 v3	1	0.75	0.75	20	SSD	✓	✓		✓	✓
StdA0-2660	StandardA0	East Japan	Intel Xeon E5-2660	1	0.75	0.75	20	SSD	✓	✓		✓	✓
StdA1-2673v3	StandardA1	Central Canada	Intel Xeon E5-2673 v3	1	1.75	1.75	70	SSD	✓	✓	✓	✓	✓
StdA1-2660	StandardA1	East Japan	Intel Xeon E5-2660	1	1.75	1.75	70	SSD	✓	✓		✓	✓
StdA1v2-2660	StandardA1 v2	South Central US	Intel Xeon E5-2660	1	2	2	10	SSD	✓		✓	✓	✓
StdA1v2-2673v3	StandardA1 v2	West Central US	Intel Xeon E5-2673 v3	1	2	2	10	SSD	✓	✓	✓	✓	✓
StdA2-2673v3	StandardA2	Central Canada	Intel Xeon E5-2673 v3	2	3.5	1.75	135	SSD	✓	✓	✓	✓	✓
StdA2-2660	StandardA2	East Japan	Intel Xeon E5-2660	2	3.5	1.75	135	SSD	✓	✓	✓	✓	✓
StdA2v2-2660	StandardA2 v2	South Central US	Intel Xeon E5-2673 v3	2	4	2	20	SSD	✓		✓	✓	✓
StdA2v2-2673v3	StandardA2 v2	West Central US	Intel Xeon E5-2673 v3	2	4	2	20	SSD	✓	✓	✓	✓	✓
StdD1-2660	StandardD1	South East Australia	Intel Xeon E5-2660	1	3.5	3.5	50	SSD	✓	✓	✓	✓	✓
StdD1v2-2673v3	StandardD1 v2	Central US	Intel Xeon E5-2673 v3	1	3.5	3.5	50	SSD	✓	✓	✓	✓	✓
StdD1v2-2673v4	StandardD1 v2	South India	Intel Xeon E5-2673 v4	1	3.5	3.5	50	SSD	✓				
StdD2-2660	StandardD2	South East Australia	Intel Xeon E5-2660	2	7	3.5	100	SSD	✓	✓	✓	✓	✓
StdD2v2-2673v3	StandardD2 v2	Central US	Intel Xeon E5-2673 v3	2	7	3.5	100	SSD	✓	✓	✓	✓	✓
StdD2v2-2673v4	StandardD2 v2	South India	Intel Xeon E5-2673 v4	2	7	3.5	100	SSD	✓				
StdD2v3-2673v4	StandardD2 v3	South East Australia	Intel Xeon E5-2673 v4	2	8	4	50	SSD	✓	✓	✓	✓	✓
StdD2v3-2673v3	StandardD2 v3	South East Asia	Intel Xeon E5-2673 v3	2	8	4	50	SSD	✓	✓	✓	✓	✓
StdE2v3-2673v4	StandardE2 v3	West Europe	Intel Xeon E5-2673 v4	2	16	8	50	SSD	✓	✓	✓	✓	✓
StdF1-2673v3	StandardF1	East US	Intel Xeon E5-2673 v3	1	2	2	16	SSD	✓	✓	✓	✓	✓
StdF1-2673v4	StandardF1	South India	Intel Xeon E5-2673 v4	1	2	2	16	SSD	✓				
StdF2-2673v3	StandardF2	East US	Intel Xeon E5-2673 v3	2	4	2	32	SSD	✓	✓	✓	✓	✓
StdF2-2673v4	StandardF2	South India	Intel Xeon E5-2673 v4	2	4	2	32	SSD	✓				
StdF2sv2-8168	StandardF2 v2	West US 2	Intel Xeon Platinum 8168	2	4	2	16	SSD	✓	✓	✓	✓	✓
StdG1-2698Bv3	StandardG1	East US 2	Intel Xeon E5-2698B v3	2	28	14	384	SSD	✓	✓	✓	✓	✓

RQ1 How accurate are properties' prediction models? (Section 4.3.2)

RQ2 How accurate are properties' transferring models? (Section 4.3.2)

RQ3 How accurate are approximated Pareto frontiers $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ compared to actual Pareto frontiers $\mathbb{C}^{PF}_{h_{src}}$ on h_{src}? (Section 4.3.2)

RQ4 How accurate are transferred Pareto frontiers $\widehat{\mathbb{C}}^{PF}_{h_{dst}}$ compared to actual Pareto frontiers $\mathbb{C}^{PF}_{h_{dst}}$ on h_{dst}? (Section 4.3.2)

4.3.1 Experimental Setup

Subject Hardware Environments

To enhance external validity of our work, we had to perform our experiments on a wide variety of hardware environments. Moreover, we wanted to run our experiments on real-world production

Table 4.2: Summary of all CPUs used in the experiment; TCH – technology node (nm), FRQ – CPU core frequency (MHz), FBS – front-side bus frequency (MHz), CM – clock multiplier, CRS – number of CPU cores, TRD – number of threads, L1i – Level 1 instruction cache size (KB), L1d – Level 1 data cache size (KB), L2 – Level 2 cache size, L3 – Level 3 cache size, PMC – PassMark CPU benchmark score (higher is better), PMT – PassMark single thread benchmark score (higher is better), PMR – PassMark overall CPU rank in PassMark database (lower is better)

General		Architecture							Caches per CPU				Caches per core			PassMark Scores		
CPU Model Name	**First Seen**	**Type**	**TCH**	**FRQ**	**FBS**	**CM**	**CRS**	**TRD**	**L1i**	**L1d**	**L2**	**L3**	**L1i**	**L1d**	**L2**	**PMC**	**PMT**	**PMR**
AMD Opteron 4171 HE	Q4 2010	K10	45	2100	3200		6	6	384	384	3072	6144	64	64	512	**3664**[1]	**732**[1]	**1147**[1]
Intel Xeon E5-2660	Q2 2012	Sandy Bridge	32	2200	4000	22	8	16	256	256	2048	20480	32	32	256	**11048**	**1387**	**265**
Intel Xeon E5-2673 v3	Q2 2015	Haswell	22	2400	4800	24	12	24	384	384	3072	30720	32	32	256	**16383**	**1666**	**116**
Intel Xeon E5-2698B v3	Q2 2014	Haswell	22	2000	4800	20	16	32	512	512	4096	40960	32	32	256	**21042**[2]	**1846**[2]	**51**[2]
Intel Xeon E5-2673 v4	Q4 2016	Broadwell	14	2300	4800	23	20	40	640	640	5120	51200	32	32	256	**21474**	**1792**	**46**
Intel Xeon Platinum 8168	Q4 2017	Skylake	14	2700	5200	27	24	48	768	768	24576	33792	32	32	1024	**29131**	**2073**	**3**

[1] PassMark Scores are provided for AMD Opteron 4170 HE
[2] PassMark Scores are provided for Intel Xeon E5-2698 v3

Table 4.3: Summary of benchmarked configurable software systems; $\mathbf{N_f}$ – Number of varied system features; NC – Number of benchmarked system configurations; NS – Number of servers on which systems were benchmarked.

Name	$\mathbf{N_f}$	**NC**	**NS**
BZIP2	2	18	33
GZIP	3	36	28
XZ	4	160	27
FLAC	5	144	29
x264	8	256	30

hardware environments that could be used by other research or development teams, what would make our research even more applicable to other practitioners. We came to a conclusion that the best hardware choice for our experiments would be a public enterprise-level cloud computing solution that provides server infrastructure as a service (IaaS). Because of that, we acquired access to the Microsoft Azure cloud computing service.

Microsoft Azure is a cloud computing service that provides infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). Microsoft provided Azure Sponsorship for our team, thus allowing us to use the Azure infrastructure to run our experiments. Azure provides cloud infrastructure through a set of dedicated international data centers. After performing a thorough analysis of all virtual machines on all data centers that were available for our sponsorship, we selected 34 virtual machines that had different CPU models, RAM size, etc. We provide a summary of all selected virtual machines in Table 4.1 and a detailed summary of unique CPUs available on these machines in Table 4.2. All virtual machines ran the Linux Ubuntu Xenial 16.04 LTS operating system.

Subject Software Systems

We build, approximate, and transfer Pareto frontiers for five different software systems: BZIP2 (Seward, J. and Mena F., 2019a), GZIP (Gailly, J.-l. and Adler, M., 2016a), XZ (Collin, 2020),

Figure 4.3: Distributions of *averaged* benchmarking observations of *compression time* metric for all studied software systems and hardware environments. Each subplot represents the averaged metric distributions for a particular software. Each line represents the averaged metric distribution for a particular hardware.

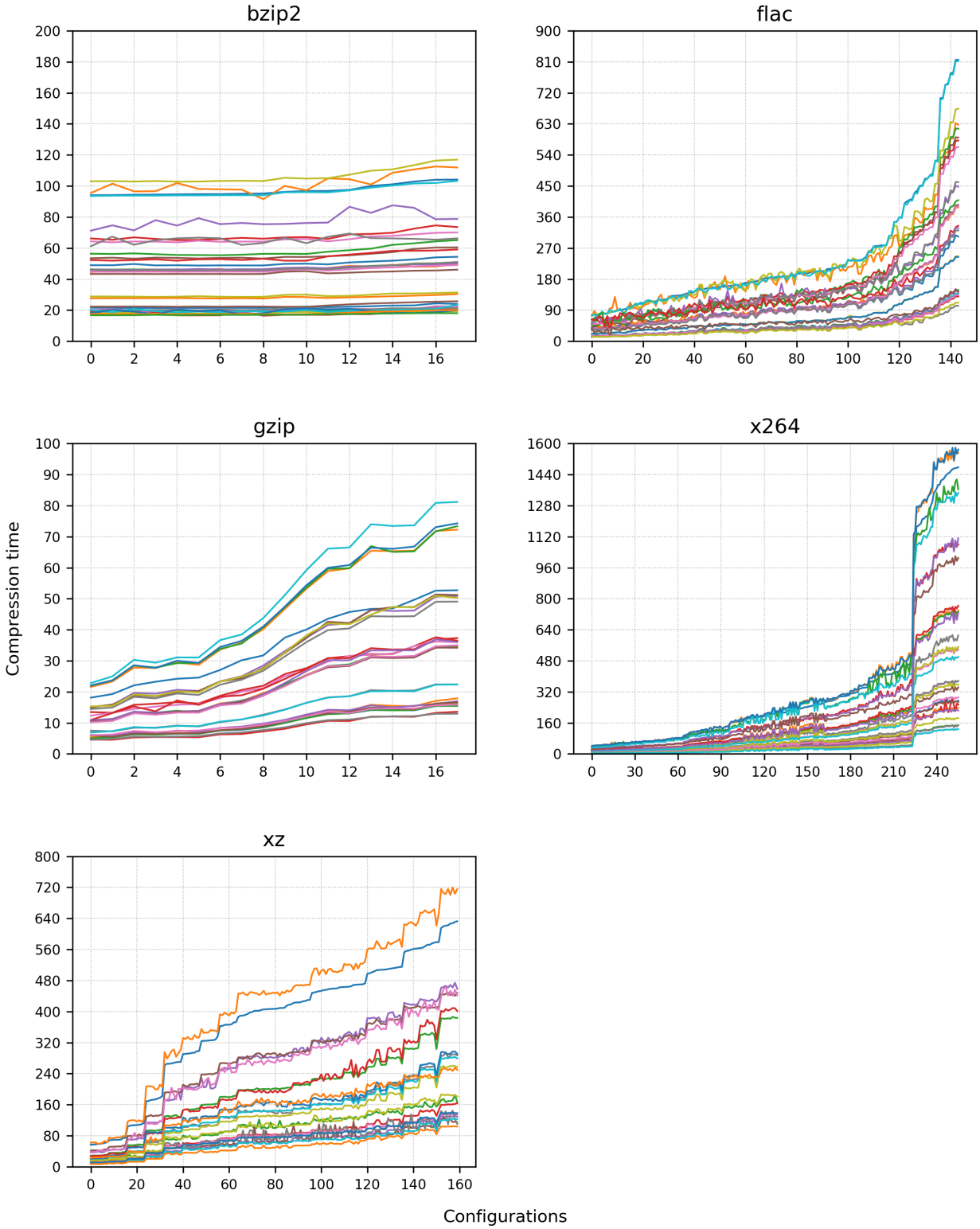


Figure 4.4: Distributions of benchmarking observations of *compression time* metric for *bzip2* software system. Each subplot represents the metric distributions on a particular hardware. Each boxplot represents the metric distribution for a particular software configuration.

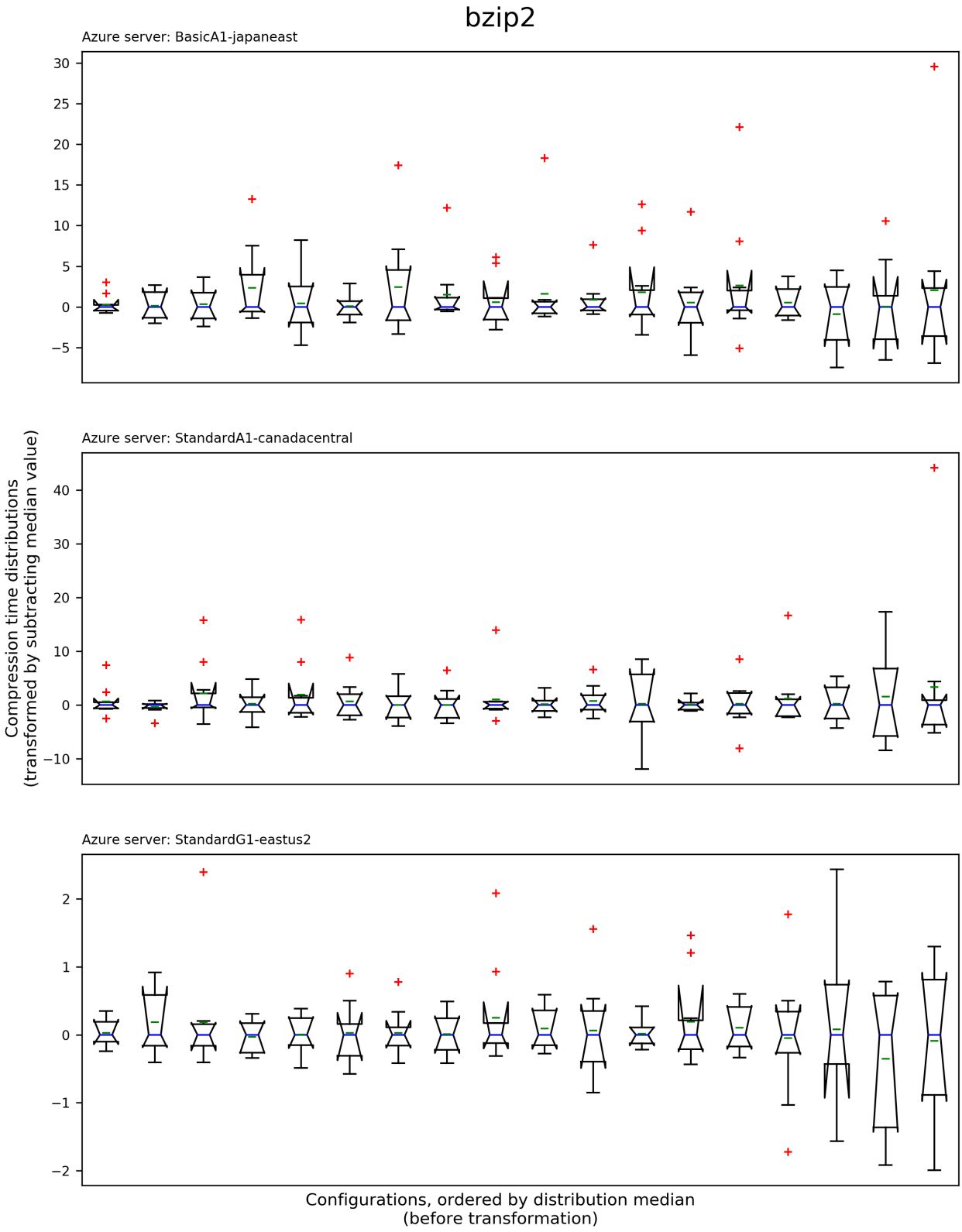


Figure 4.5: Distributions of benchmarking observations of *compression time* metric for *flac* software system. Each subplot represents the metric distributions on a particular hardware. Each boxplot represents the metric distribution for a particular software configuration.

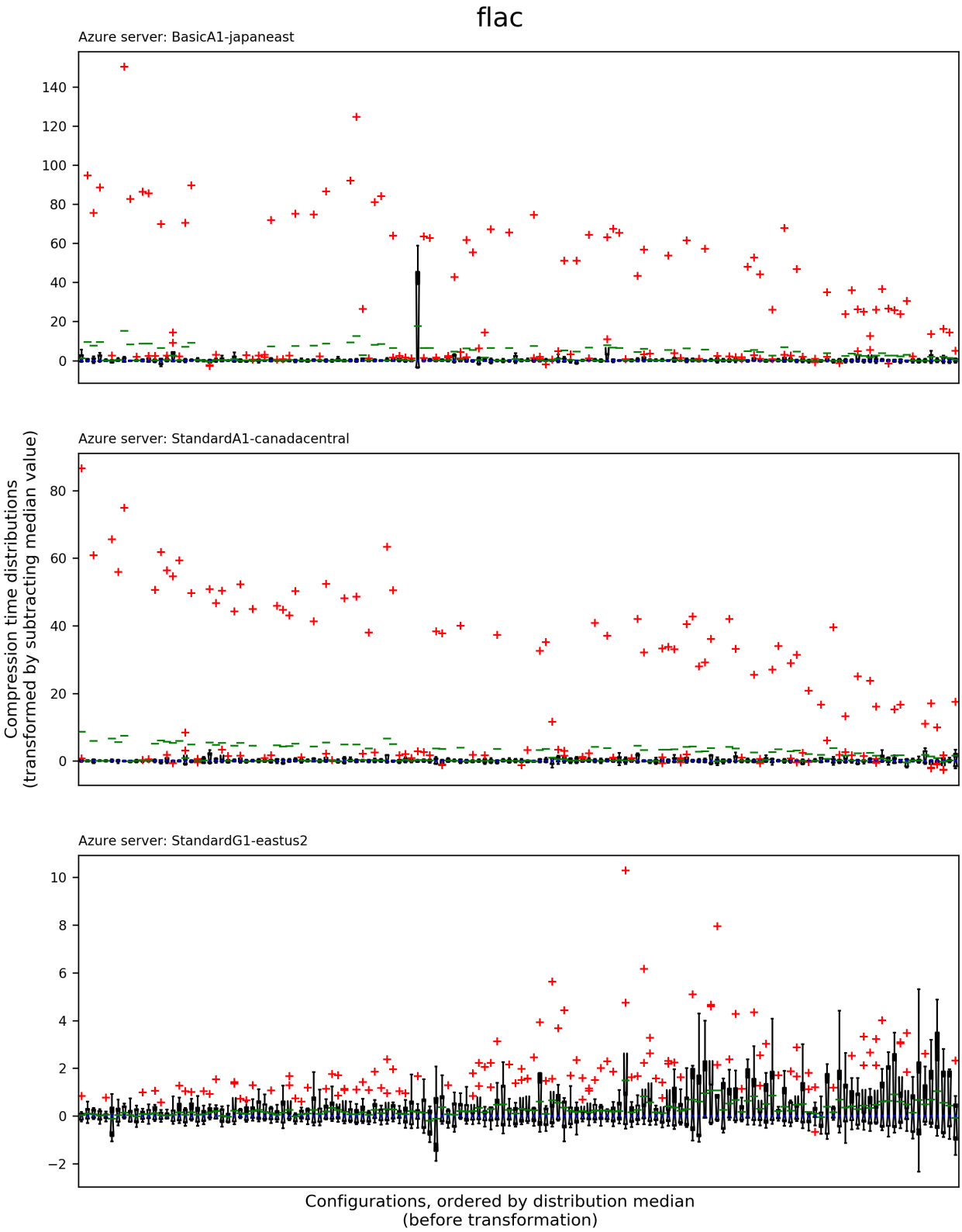


Figure 4.6: Distributions of benchmarking observations of *compression time* metric for *gzip* software system. Each subplot represents the metric distributions on a particular hardware. Each boxplot represents the metric distribution for a particular software configuration.

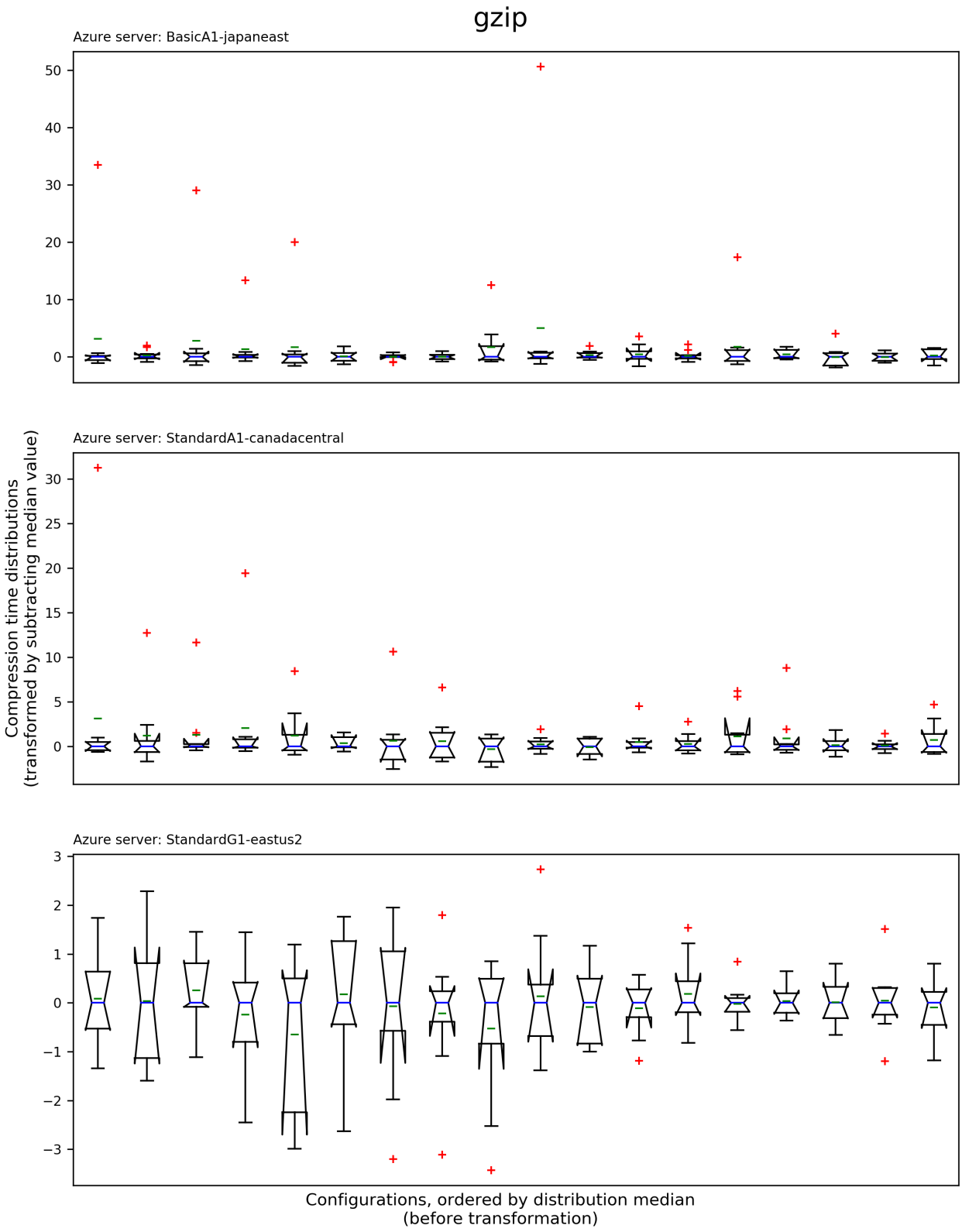


Figure 4.7: Distributions of benchmarking observations of *compression time* metric for *x264* software system. Each subplot represents the metric distributions on a particular hardware. Each boxplot represents the metric distribution for a particular software configuration.

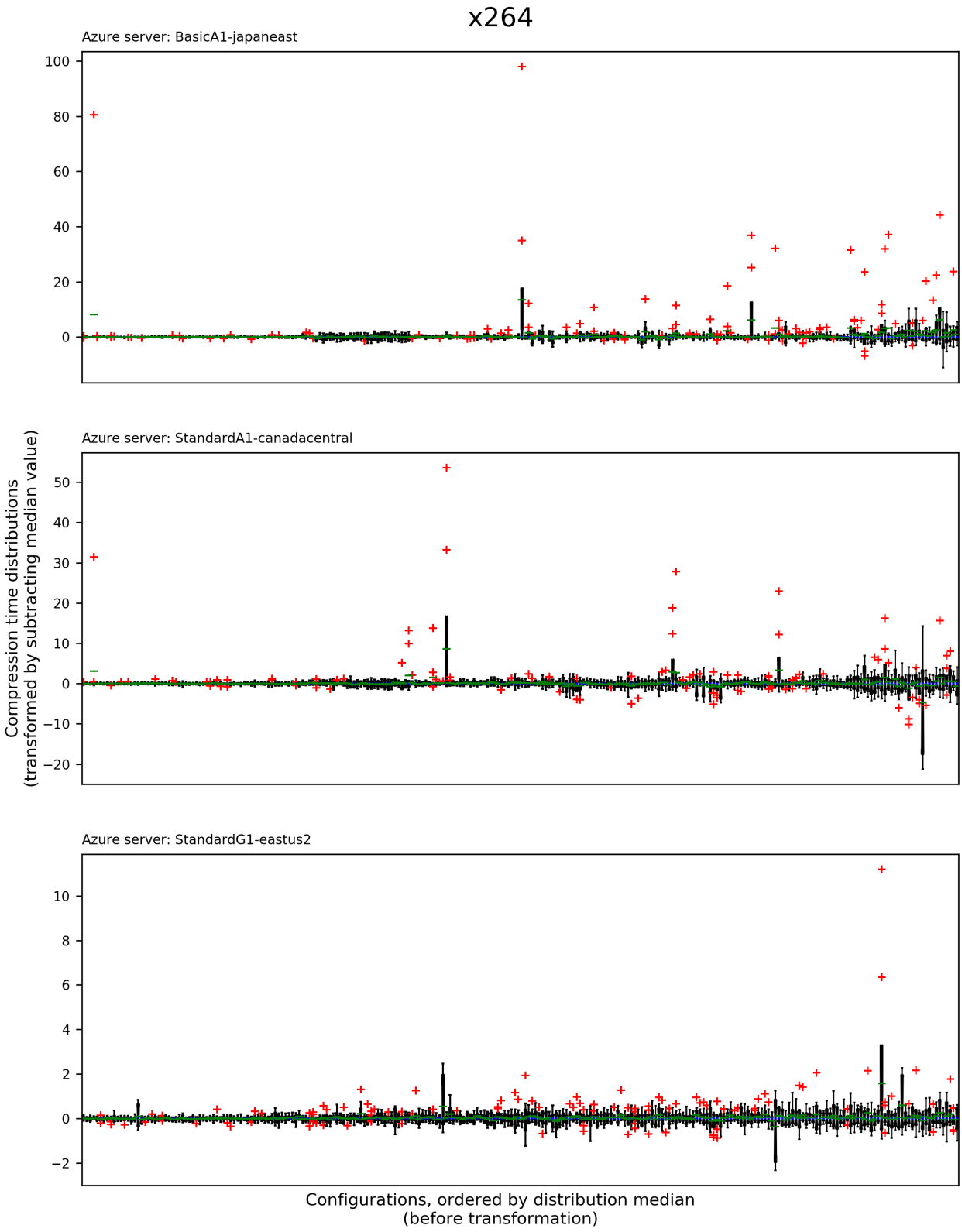
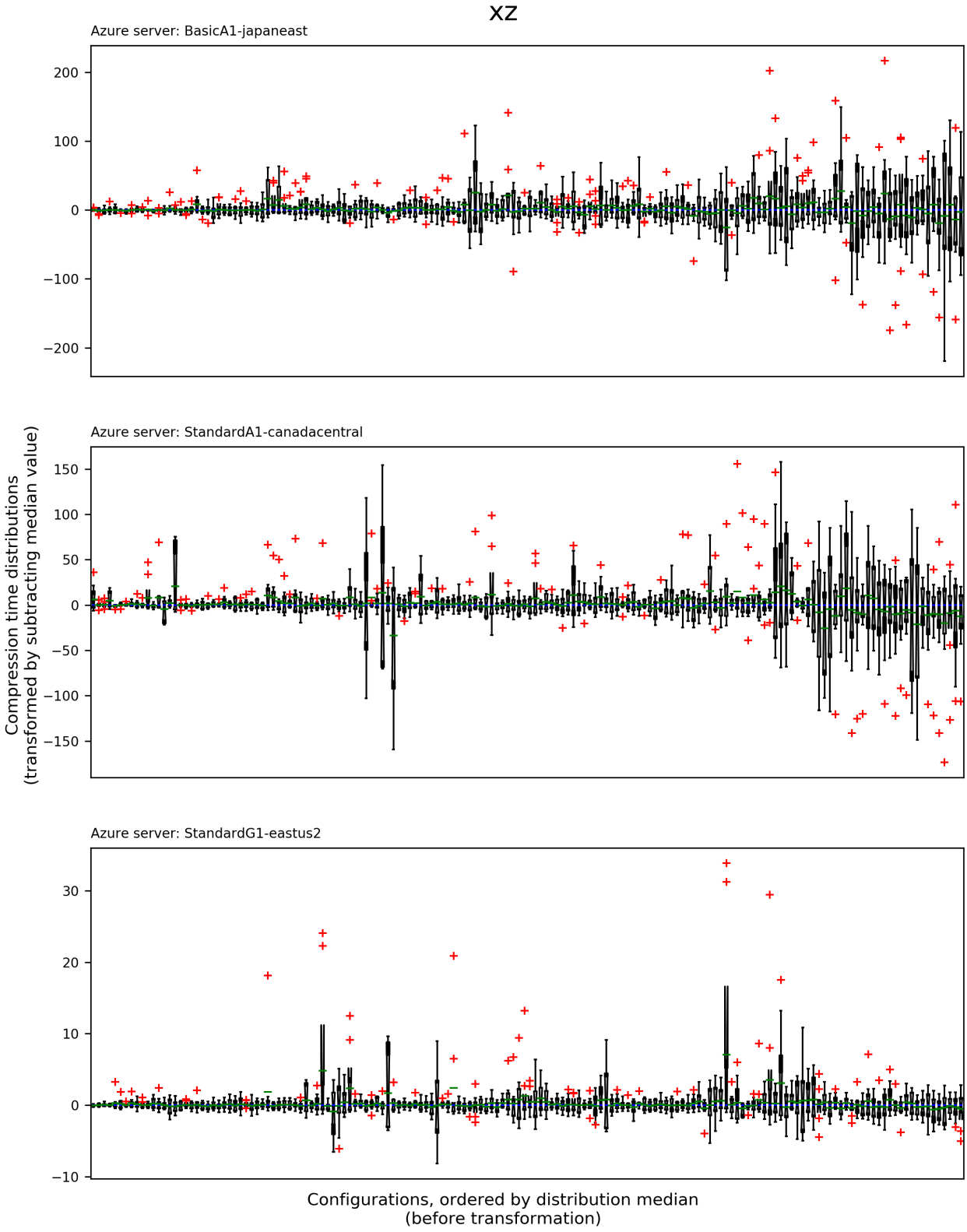


Figure 4.8: Distributions of benchmarking observations of *compression time* metric for *xz* software system. Each subplot represents the metric distributions on a particular hardware. Each boxplot represents the metric distribution for a particular software configuration.



FLAC (Josh Coalson, Xiph.Org Foundation, 2019a), x264 (VideoLAN Organization, 2020a). BZIP2 is a general-purpose data compression program which utilizes the Burrows-Wheeler algorithm. GZIP is a general-purpose data compression utility which employs the DEFLATE lossless compression algorithm. XZ is a multi-threaded general-purpose data compression software which uses the LZMA2 lossless compression algorithm. FLAC (Free Lossless Audio Codec) is an audio compression software that uses homonymous lossless audio coding format. x264 is a video encoding software that uses lossy H.264/MPEG-4 AVC format. We focus on systems from compression and multimedia transformation domains, since these systems are intuitive, have a large userbase, and provide a variety of features. Moreover, these systems share common properties, what allows straightforward comparison of how well our approach works for different use cases.

We performed comprehensive benchmarking of each selected software system. For each software system we selected a benchmark in order to test the generated configurations upon. We benchmarked BZIP2, GZIP, and XZ using the *large text compression benchmark* (Mahoney, M., 2019), which represents first 10^9 bytes of the Wikipedia XML archive. We benchmarked FLAC using *Ghosts I-IV* (Nine Inch Nails, 2008), a music album of a band 'Nine Inch Nails', that contains 36 tracks of improvisation music, released under the Creative Commons license. We benchmarked x264 using a trailer of *Sintel* (Levy, C., 2010), an open-content film created and released under the Creative Commons license by Blender Foundation.

For each software system we selected a set of configurable features that had demonstrated the strongest influence on studied systems' metrics during our preliminary experiments. We vary two different BZIP2 features: *n* and *small* (see Seward, J. and Mena F., 2019b, for description). We vary three different GZIP features: *n*, *rsyncable*, and *synchronous* (Gailly, J.-l. and Adler, M., 2016b). We vary four different XZ features: *n*, *no-sparse*, *extreme*, *check* (Collin, L., 2020). We vary five different FLAC features: *n*, *verify*, *lax*, *replay-gain*, and *p* (Josh Coalson, Xiph.Org Foundation, 2019b). We analyze eight different X264 features: *b-adapt*, *me*, *no-mbtree*, *no-scenecut*, *rc-lookahead*, *ref*, *subme*, and *trellis* (VideoLAN Organization, 2020b).

Using the selected features, we generated a set of all possible valid configurations for each system. To improve internal validity, we measured each configuration on all hardware environments 10 times, which was the largest number that our Azure budget allowed.

For each configuration we measured two properties: *compression time* and *compressed size*. We selected these properties because they are intuitive, easy to measure, and are universal to all studied systems. It is worth to notice that the *compressed size* property doesn't change it's value for a single configuration across hardware. Nevertheless, since our approach works using small samples of measured configurations, it has to approximate this property for the whole configuration space. Therefore, the *compressed size* property still has a strong influence on approximated and transferred Pareto frontiers. We leave analysis of multiple varying properties for future work.

We present distributions of benchmarked observations of the *compression time* property for all configurations of the studied configurable software systems on Figures 4.3–4.8. Figure 4.3 presents averaged compression time property values for all configurations (ordered by their values), studied configurable software systems, and hardware environments. We can observe a similar increasing pattern for all software systems across different hardware, indicating that higher values of this property on one hardware correspond to higher values of this property on another hardware, meaning that cross-platform transferring of this property should be possible with high accuracy. Figures 4.4–4.8

present the compression time property distributions for each configuration of different configurable software systems. Figure 4.3 demonstrated that different system configurations can have property values that differ by an order of magnitude on the same hardware, therefore, in order to save space, we performed data normalization by median deduction for Figures 4.4–4.8. We can observe that the configuration distributions are generally symmetric with minimal amount of outliers, therefore, we can use an arithmetical mean for averaging the configuration distribution for further analysis.

Comprehensive benchmarking allowed us to perform and analyze the overall Pareto transferring process for each software system across all possible pairs of source and destination environments. Moreover, we exhaustively analyzed all possible training sample $\mathbb{C}_S$ sizes $[2, N_{\mathbb{C}} - 1]$ (see Section 4.2.1) and transferring sample $\mathbb{C}_{both}$ sizes $[2, N_{\mathbb{C}_{both}} - 1]$ (see Section 4.2.3). However, since the internal Azure infrastructure is being constantly updated, not all hardware environments were available for benchmarking of all software systems. Table 4.1 highlights which hardware environments we used for benchmarking of each particular software system. Table 4.2 shows specifications for each unique CPU that appeared in different hardware environments. Finally, Table 4.3 provides general information about the system benchmarking process.

4.3.2 Experimental Results

Predictors and Transferrers Accuracy

In order to answer RQ1, we performed a comprehensive evaluation of properties' predictors for all configurable software systems and hardware environments. As mentioned in Section 4.2.1, after preliminary evaluation we selected regression trees *RT* as our property prediction models, and we assessed the quality of predictors using *MAPE*, acquired using *LOOCV* validation. Figure 4.9 presents evaluation results by displaying distributions of property predictors' *MAPE* with increase of predictors' training sample size. For all software and hardware we observe a strong decreasing trend of *MAPE* with increase of training sample size. For all software and hardware regression trees could achieve a *MAPE* of less than 10%, and for majority of combinations less than 5%. This observation agrees with previous research and demonstrates that regression trees are well fit for predicting properties of configurable software systems.

Figures 4.10–4.14 present *MAPE* distributions when predicting the compression time metric using regression trees, trained using different sampling sizes. Each figure represents *MAPE* distributions for a particular software system. Each observation of a distribution corresponds to a particular experimental replication. Figure 4.9 showed that *MAPE* can have values that differ by an order of magnitude for the same software and hardware systems, therefore, in order to save space for all Figures 4.10–4.14 we performed data transformation by median deduction. We observe that *MAPE* distributions are generally symmetrical and have a minimal amount of outliers, therefore, we can use arithmetical mean for averaging *MAPE* results across different experiment replications. Moreover, we observe a strong decreasing trend in variability of *MAPE* with increase of a training sample size for all configurable software systems and hardware environments, meaning that *MAPE* becomes much more stable across different experimental replications.

In order to answer RQ2, we performed a comprehensive evaluation of properties' transferrers for all configurable software systems and hardware environments. Despite the fact that we selected

Figure 4.9: Distributions of *averaged* MAPE error of *regression-tree-based* predictors, when approximating the *compression time* metric for all studied software on heterogeneous hardware, using all possible training sample sizes. Each subplot represents MAPE distributions for a particular software. Each line represents MAPE distribution for a particular hardware.

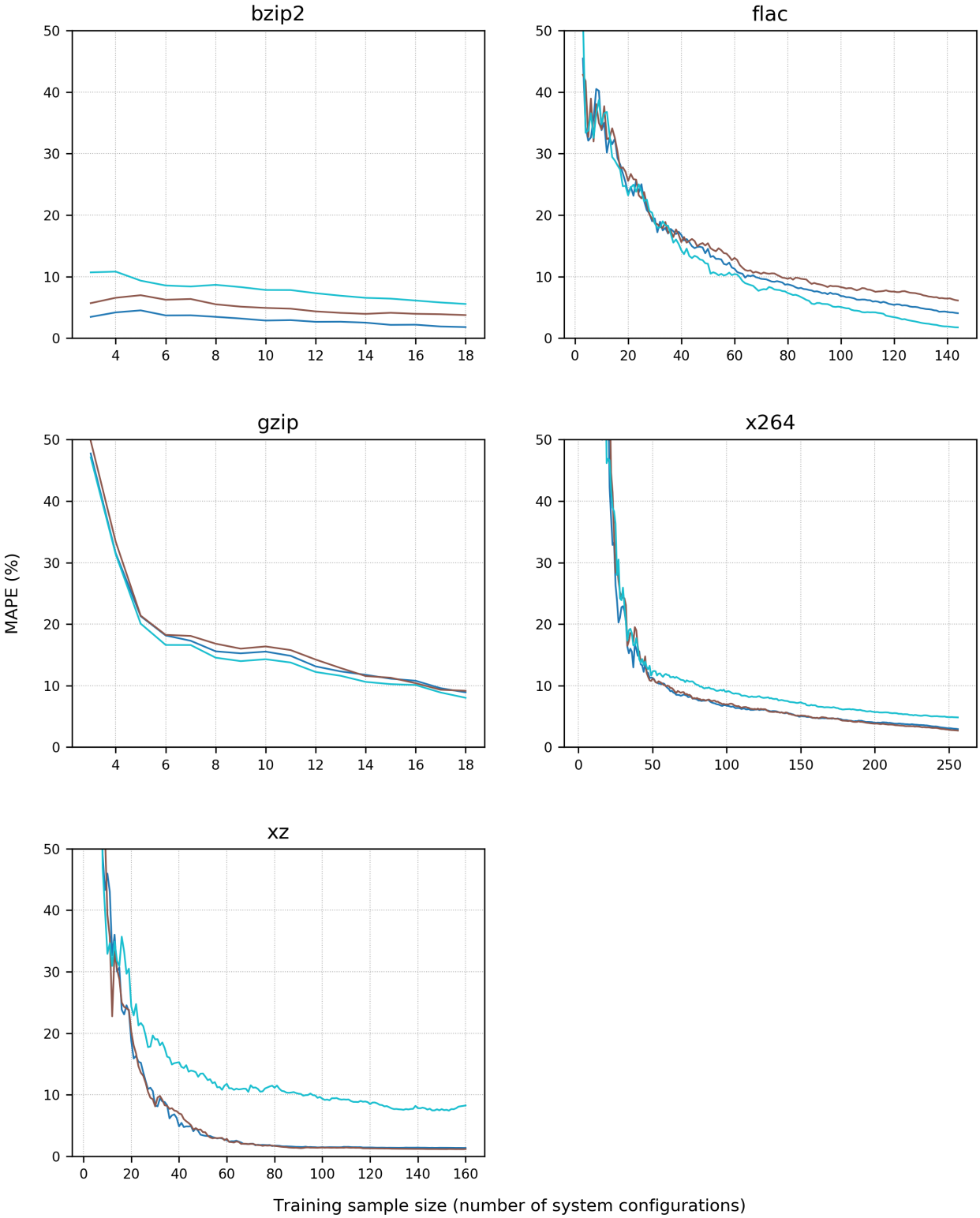


Figure 4.10: Distributions of MAPE error of *regression-tree-based* predictors, when approximating the *compression time* metric for *bzip2* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

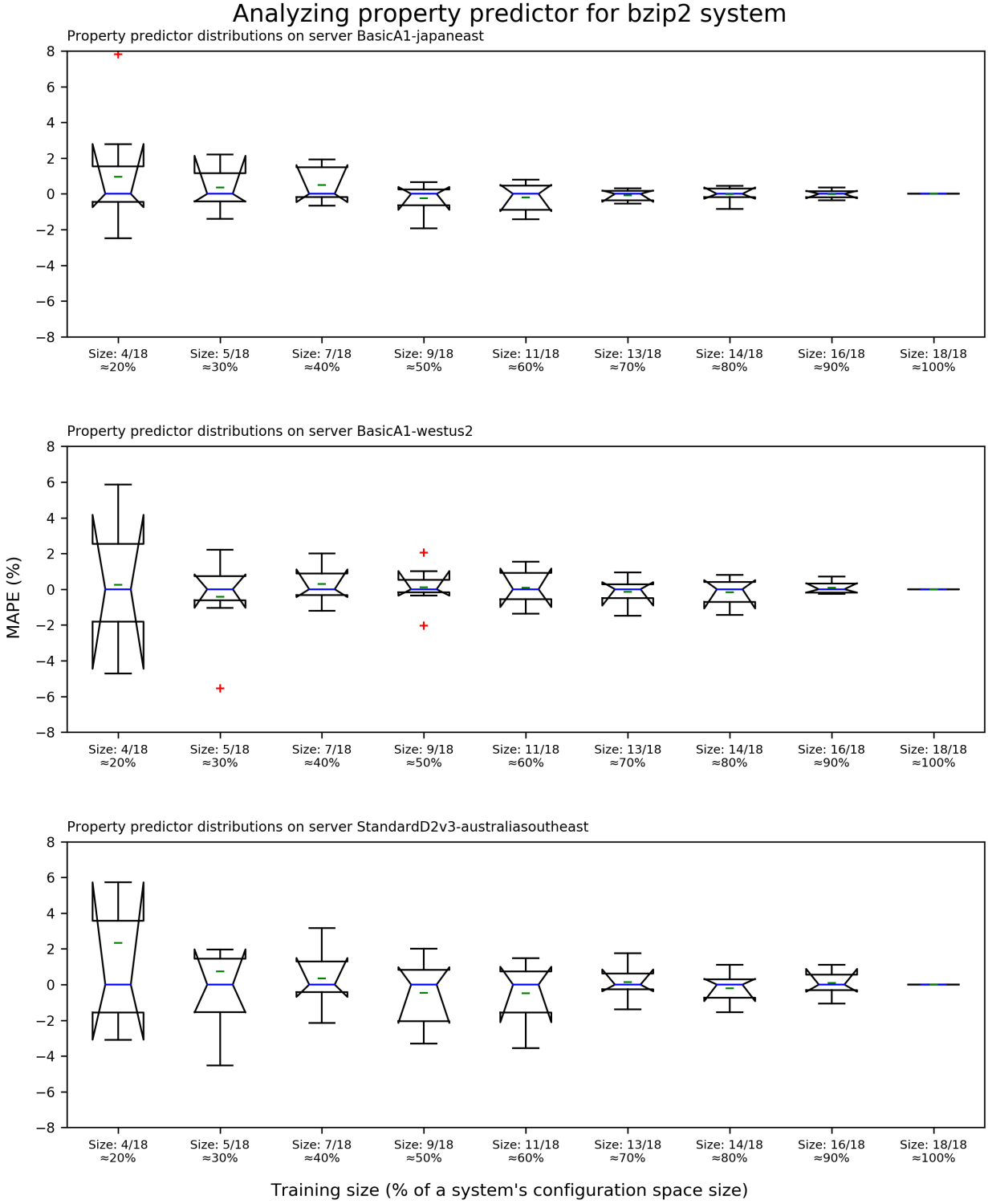


Figure 4.11: Distributions of MAPE error of *regression-tree-based* predictors, when approximating the *compression time* metric for *flac* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

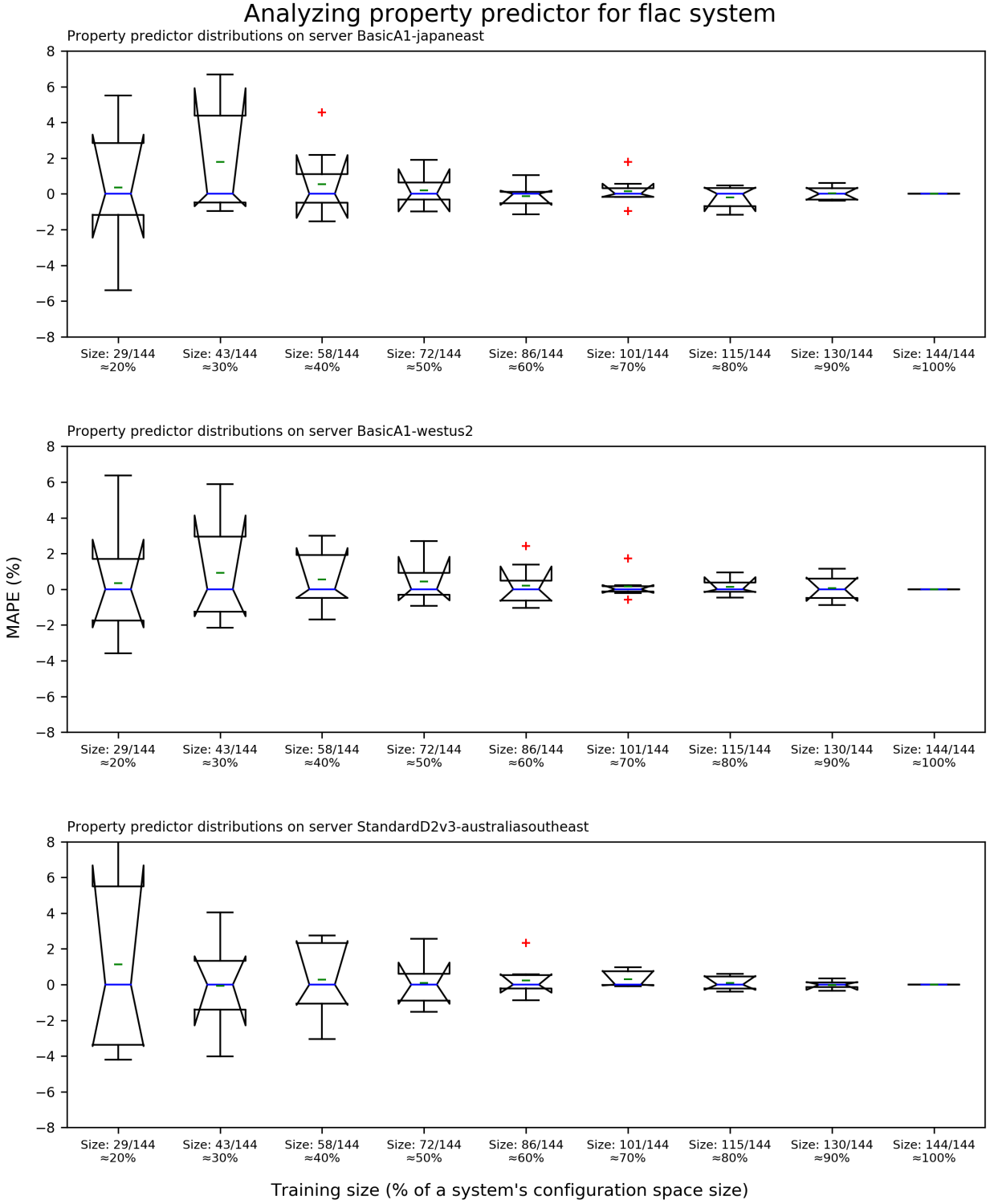


Figure 4.12: Distributions of MAPE error of *regression-tree-based* predictors, when approximating the *compression time* metric for *gzip* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

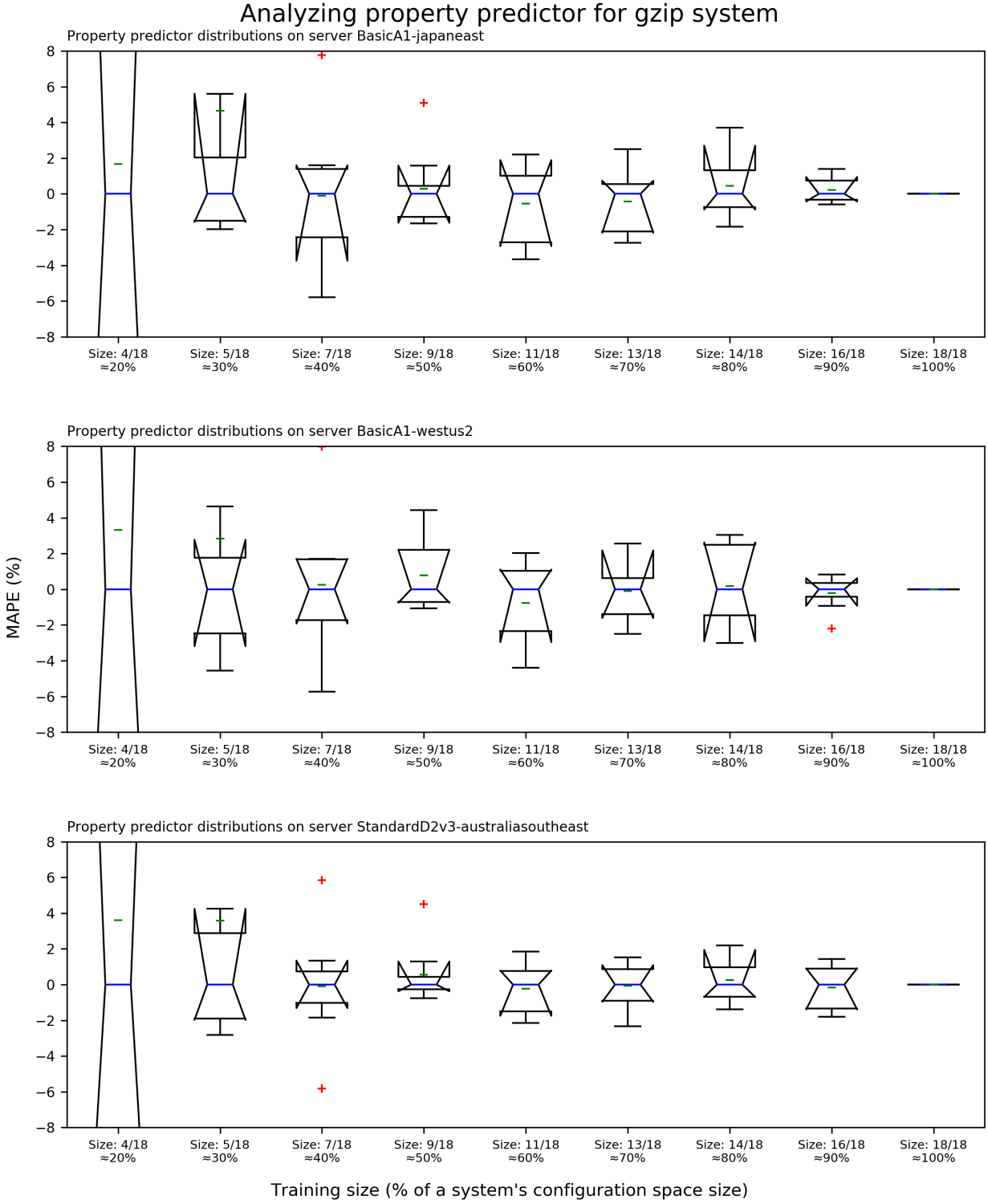


Figure 4.13: Distributions of MAPE error of *regression-tree-based* predictors, when approximating the *compression time* metric for *x264* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

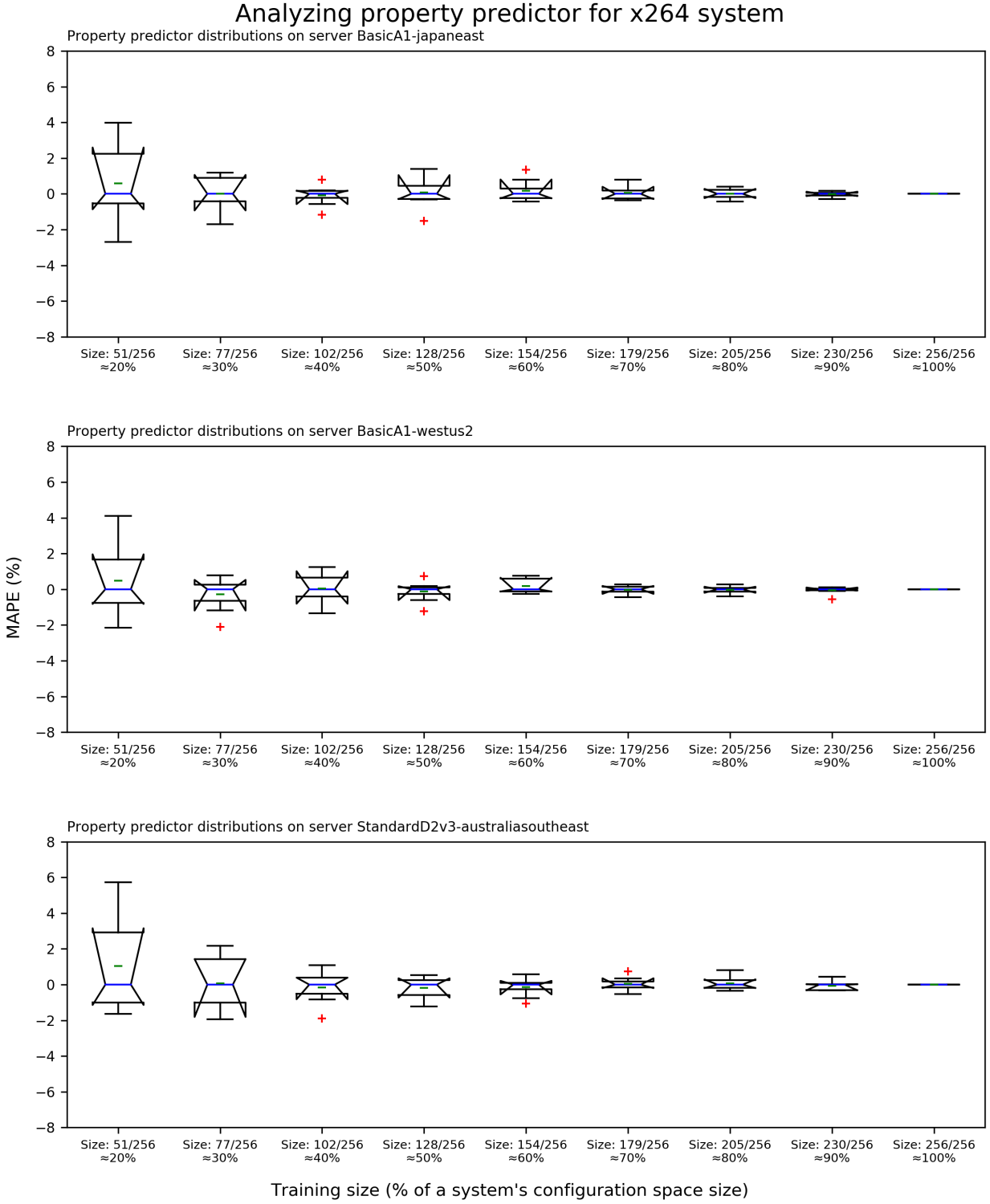


Figure 4.14: Distributions of MAPE error of *regression-tree-based* predictors, when approximating the *compression time* metric for *xz* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

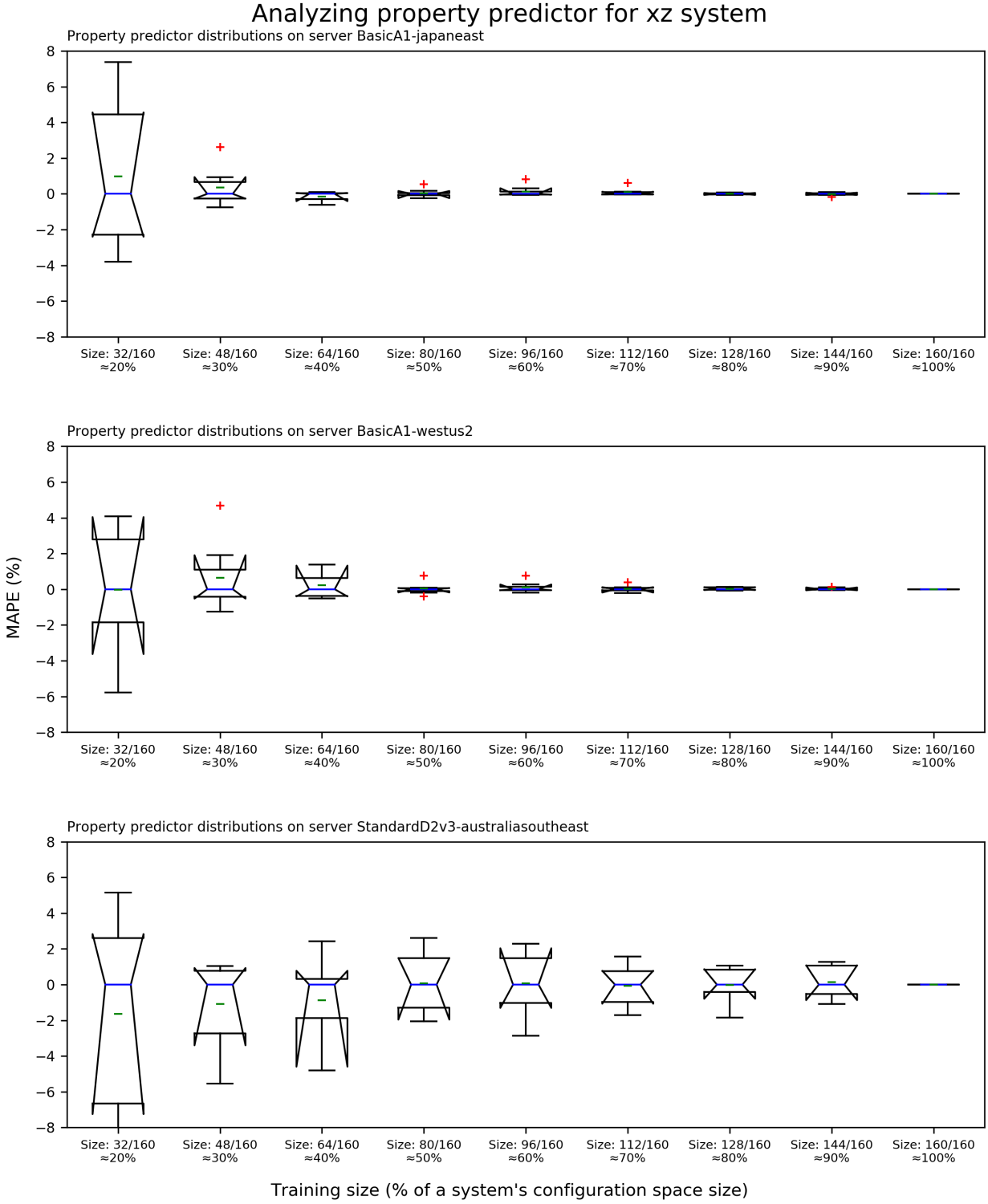


Figure 4.15: Distributions of *averaged* MAPE error of *linear-based* transferrers, when transferring the *compression time* metric across heterogeneous hardware, using all possible training sample sizes. Each subplot represents MAPE distributions for a particular software. Each line represents MAPE distribution for a particular hardware.

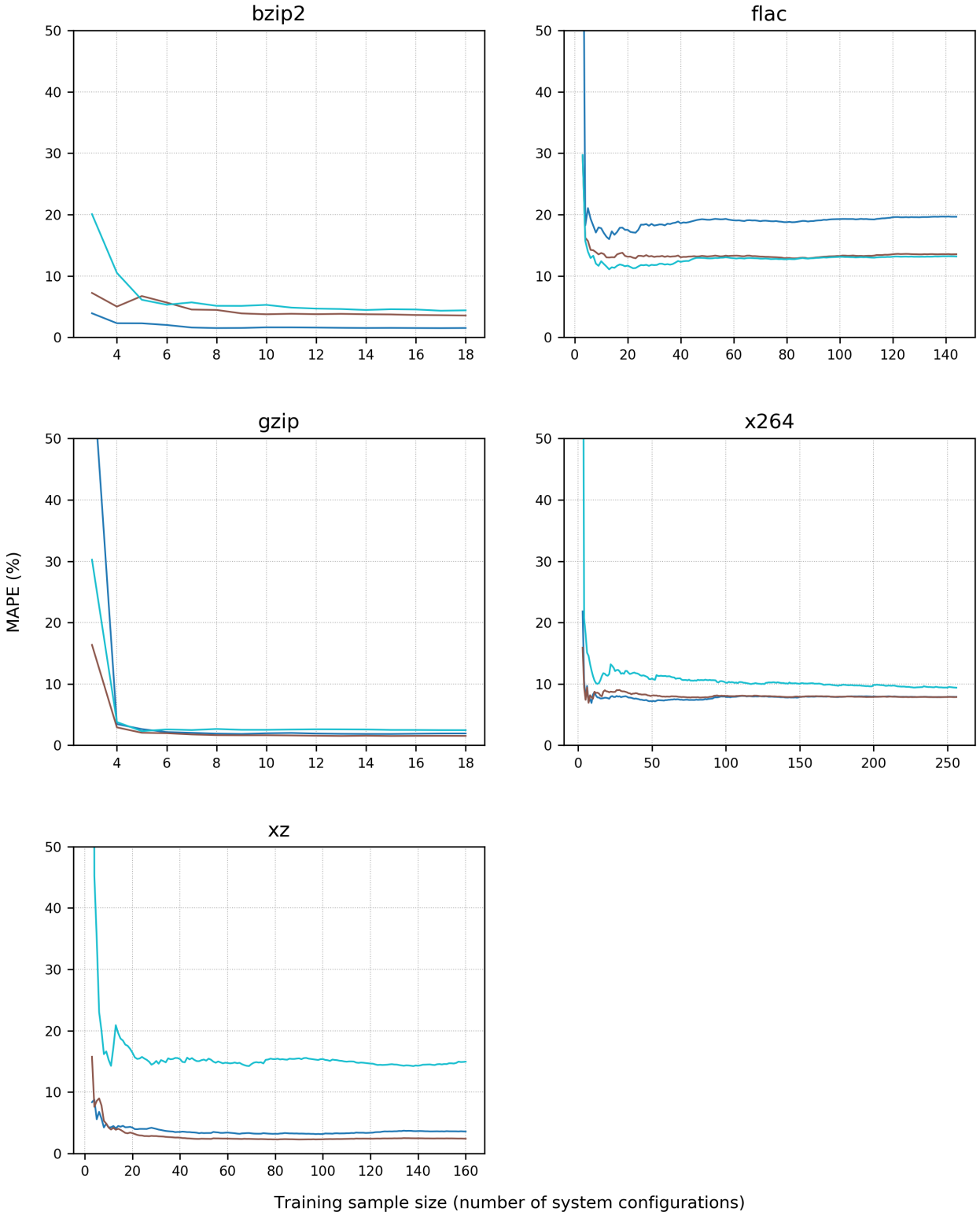


Figure 4.16: Distributions of MAPE error of *linear-based* transferrers, when transferring the *compression time* metric for *bzip2* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

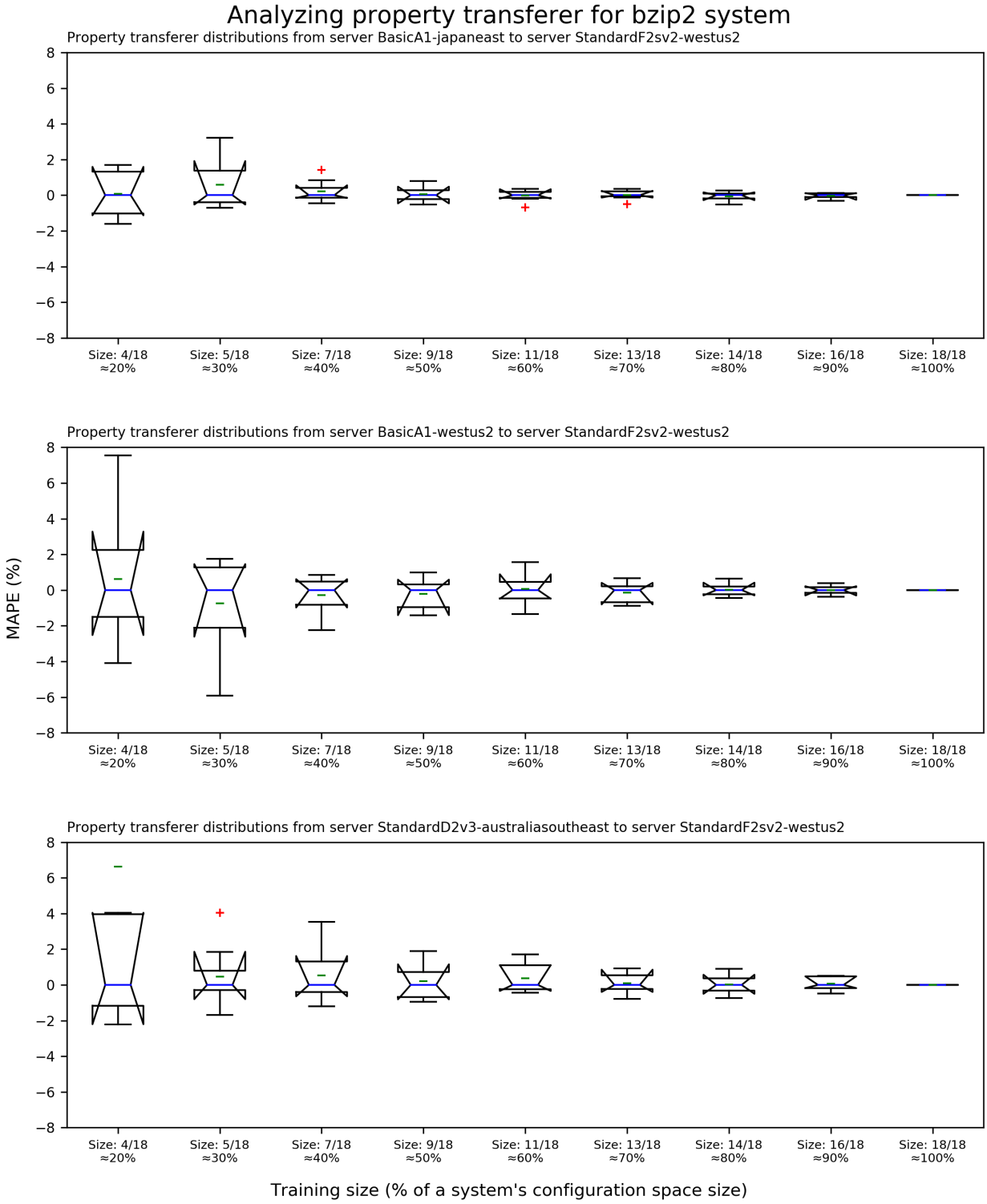


Figure 4.17: Distributions of MAPE error of *linear-based* transferrers, when transferring the *compression time* metric for *flac* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

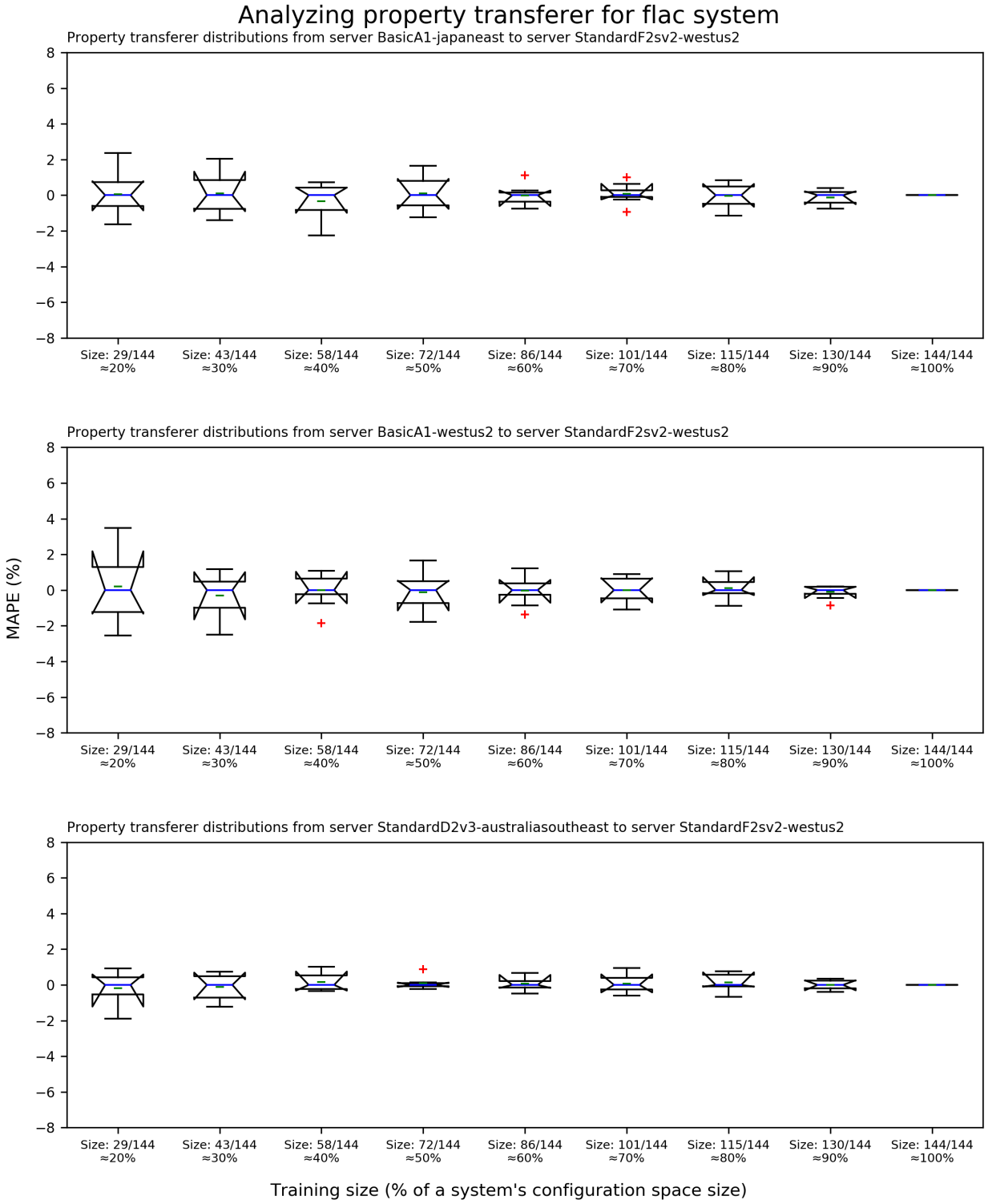


Figure 4.18: Distributions of MAPE error of *linear-based* transferrers, when transferring the *compression time* metric for *gzip* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

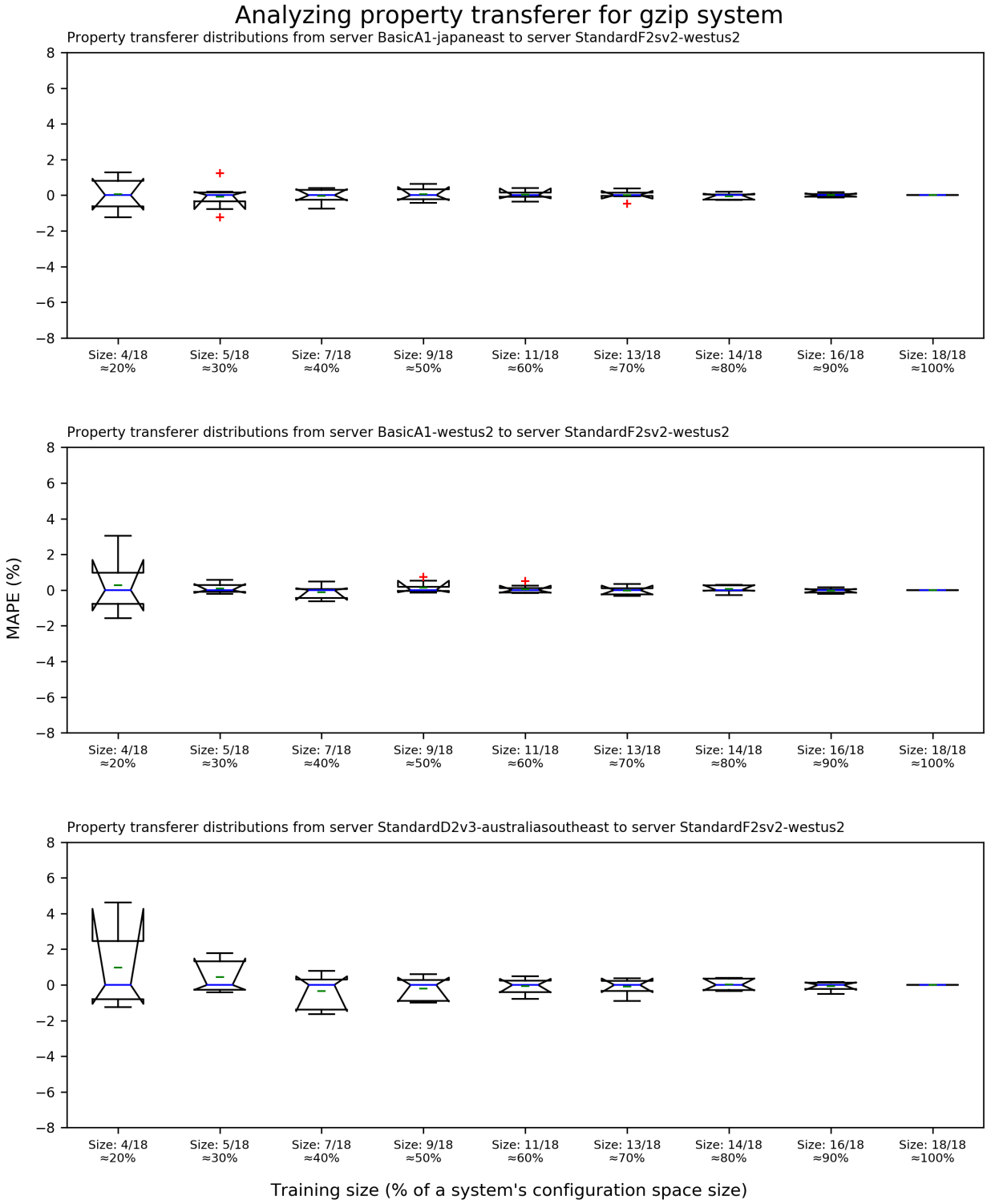


Figure 4.19: Distributions of MAPE error of *linear-based* transferrers, when transferring the *compression time* metric for *x264* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

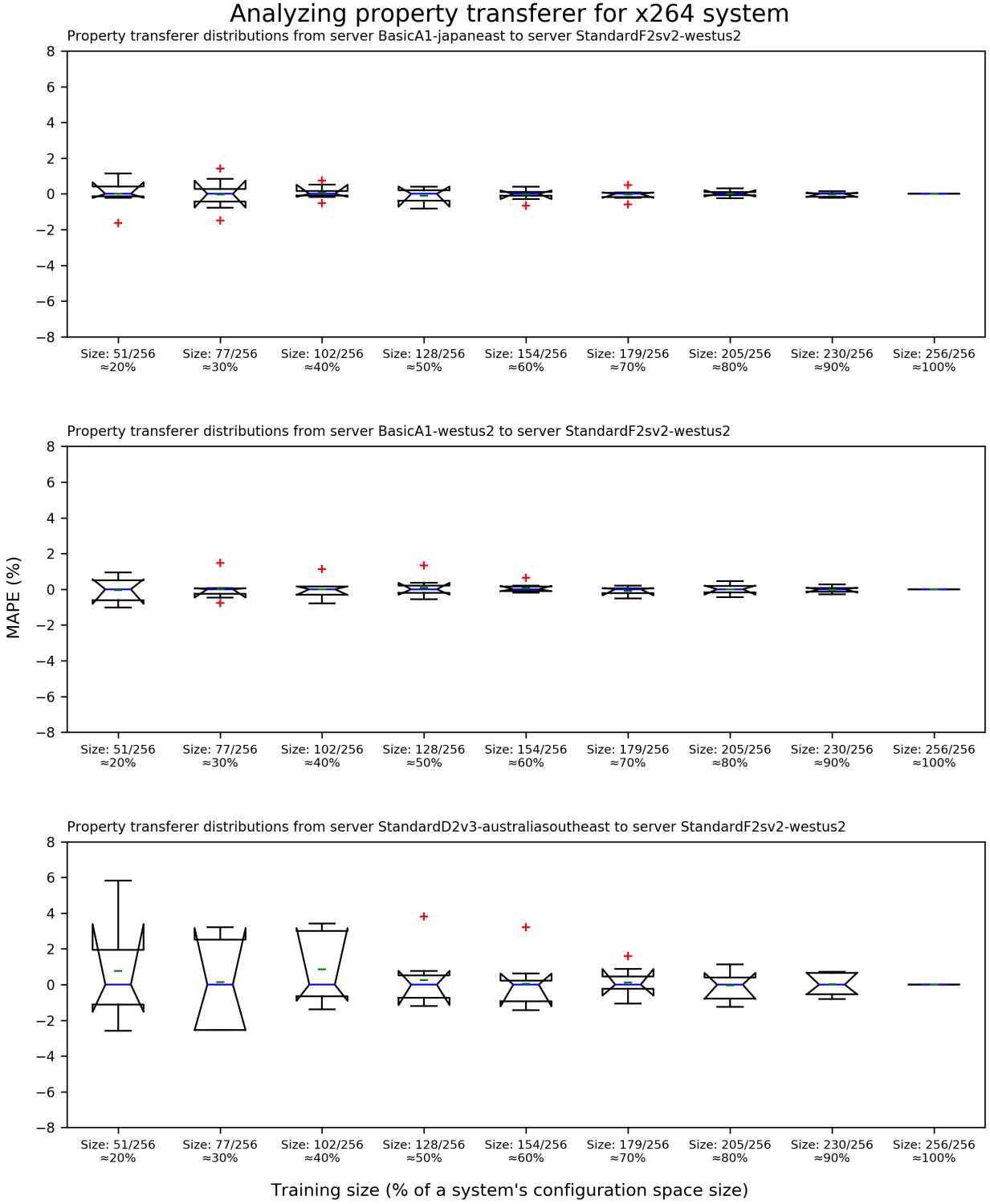


Figure 4.20: Distributions of MAPE error of *linear-based* transferrers, when transferring the *compression time* metric for *xz* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

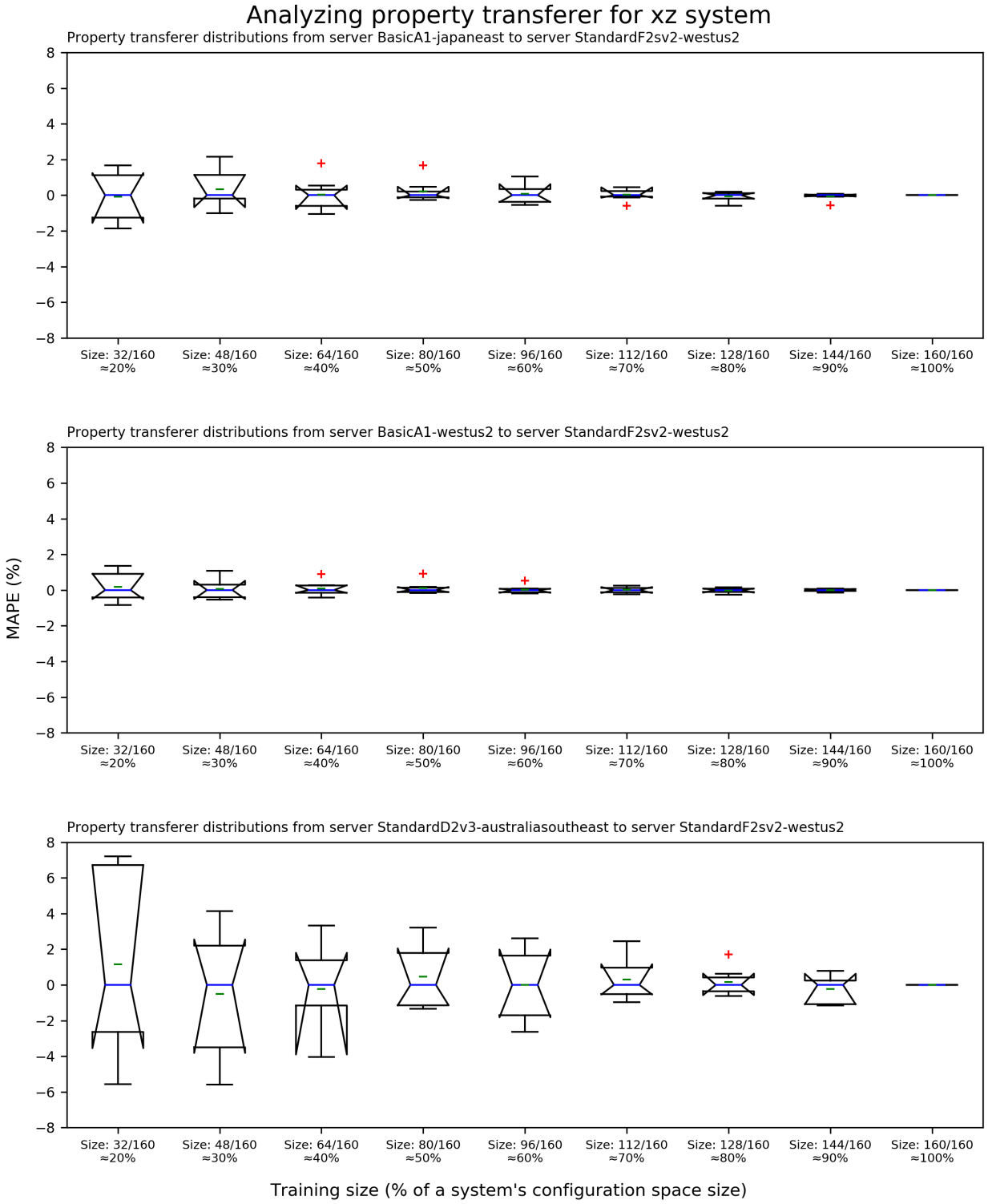


Figure 4.21: Distributions of *averaged* MAPE error of *regression-tree-based* transferrers, when transferring the *compression time* metric across heterogeneous hardware, using all possible training sample sizes. Each subplot represents MAPE distributions for a particular software. Each line represents MAPE distribution for a particular hardware.

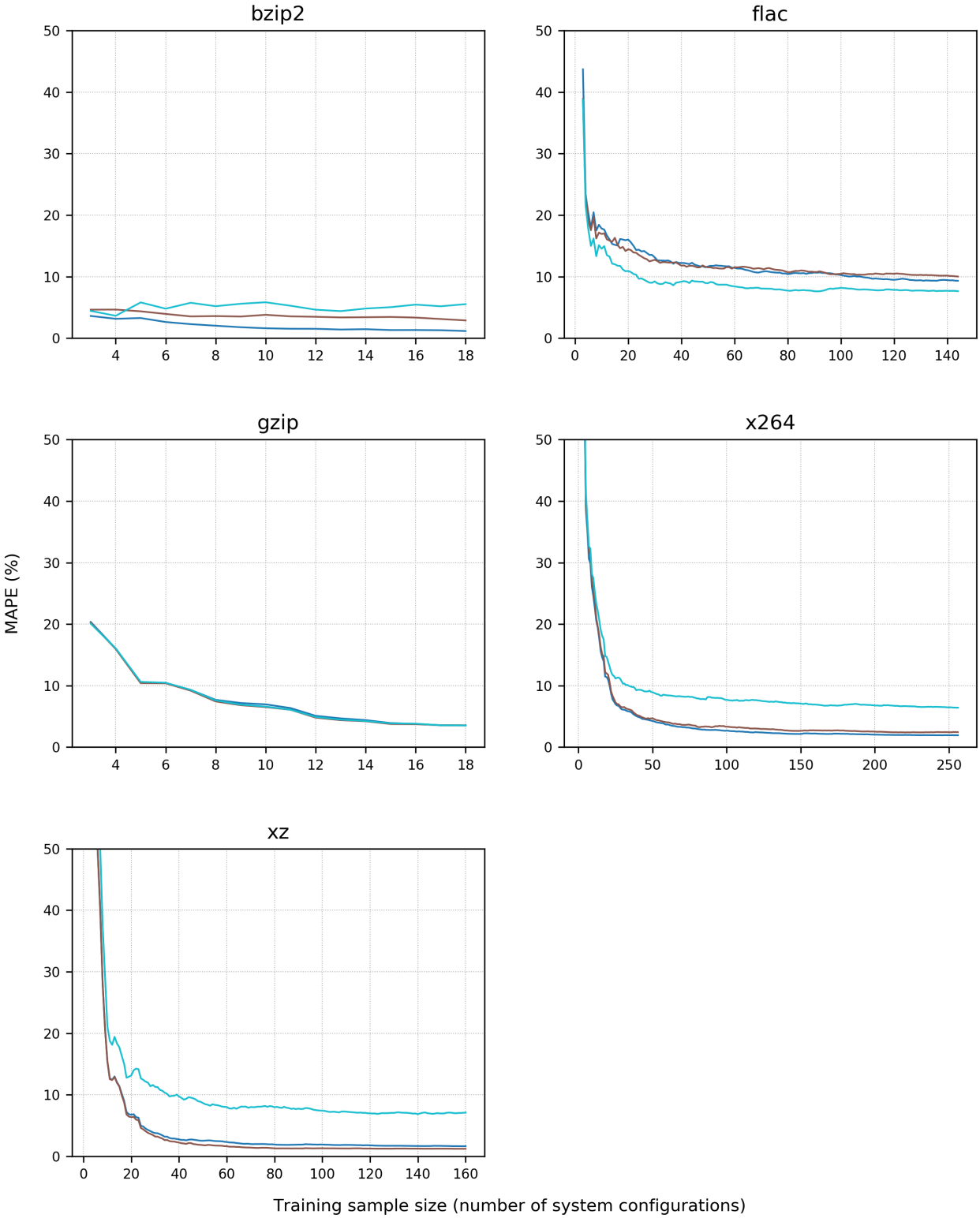


Figure 4.22: Distributions of MAPE error of *regression-trees-based* transferrers, when transferring the *compression time* metric for *bzip2* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

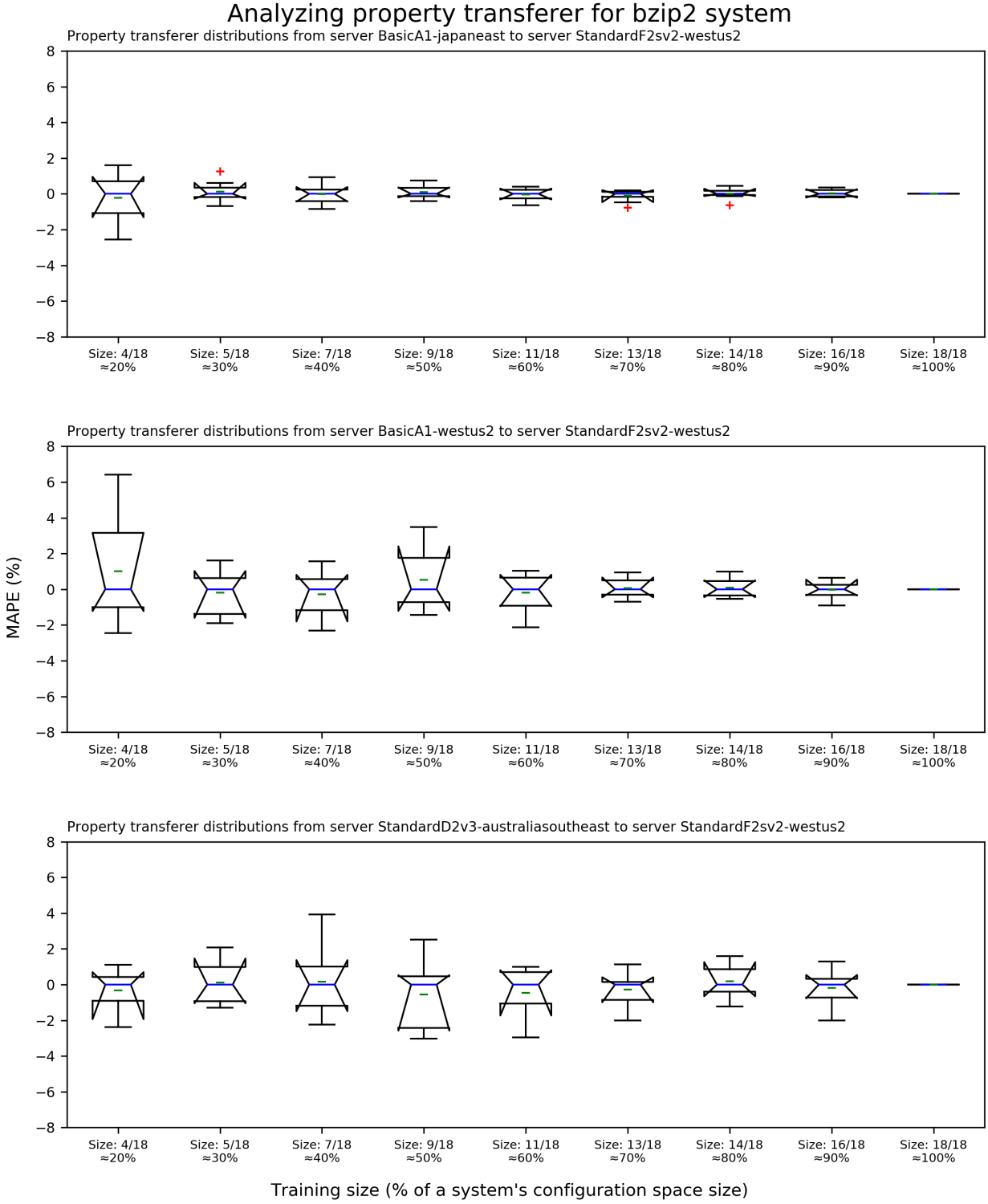


Figure 4.23: Distributions of MAPE error of *regression-trees-based* transferrers, when transferring the *compression time* metric for *flac* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

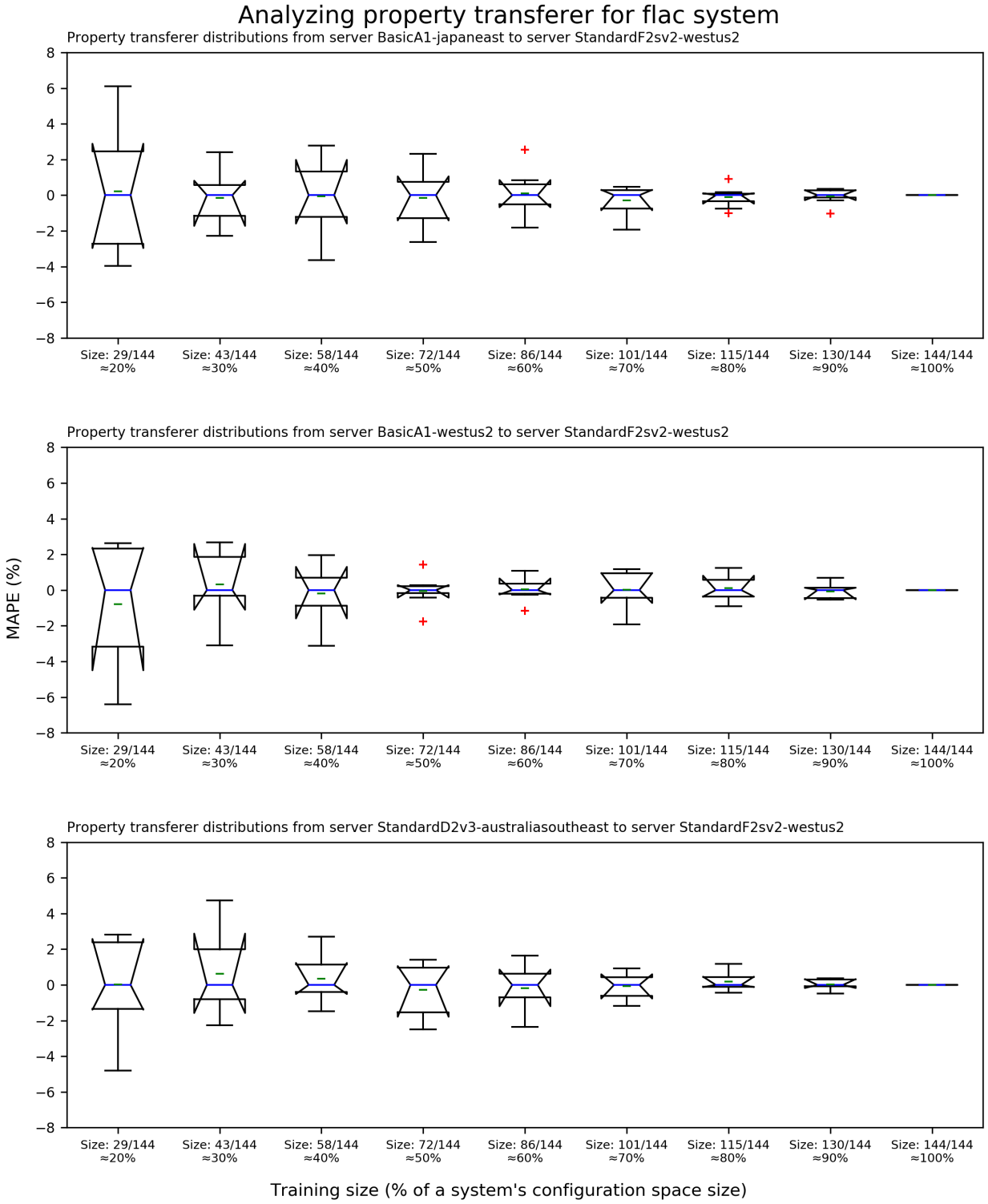


Figure 4.24: Distributions of MAPE error of *regression-trees-based* transferrers, when transferring the *compression time* metric for *gzip* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

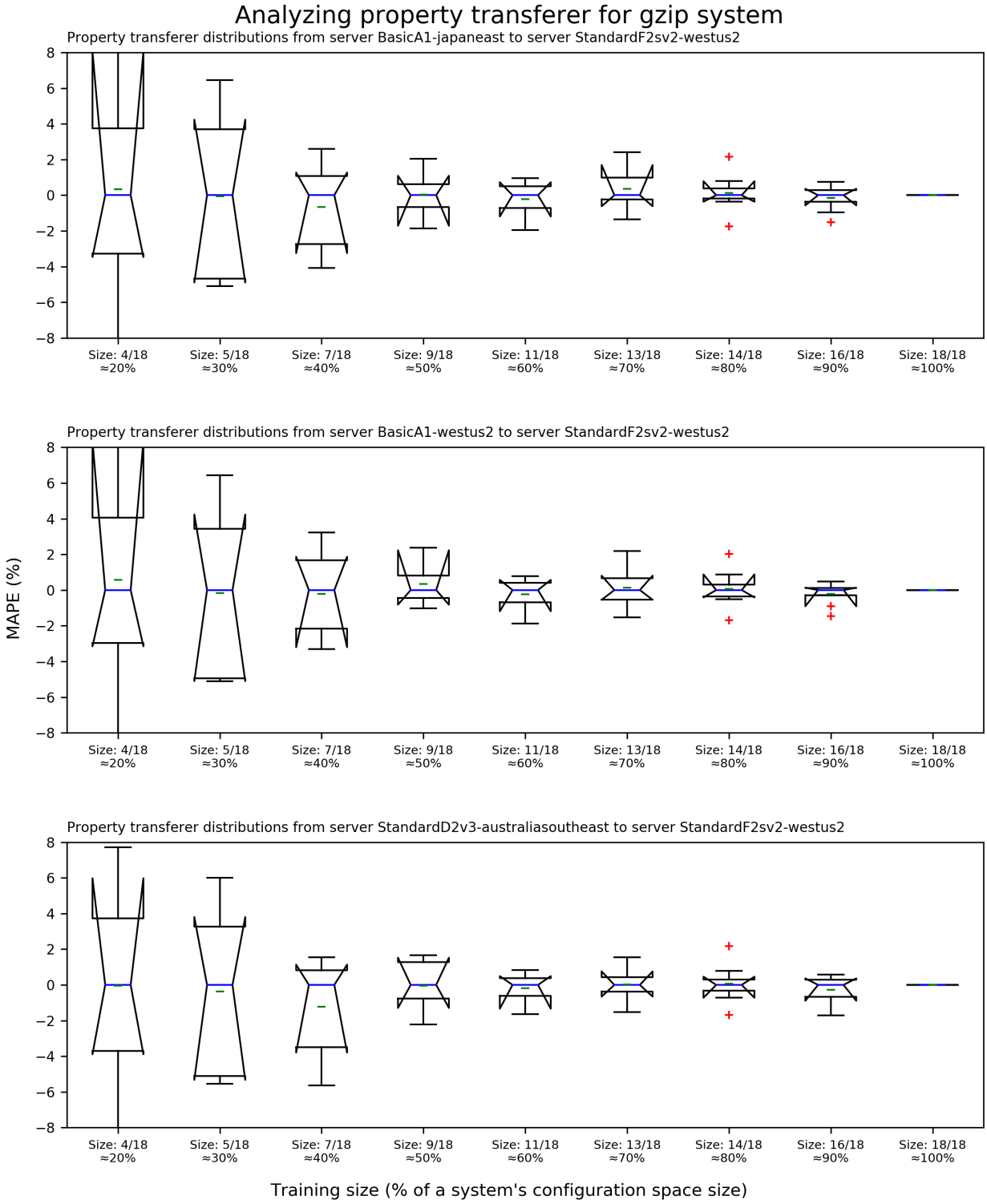


Figure 4.25: Distributions of MAPE error of *regression-trees-based* transferrers, when transferring the *compression time* metric for *x264* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.

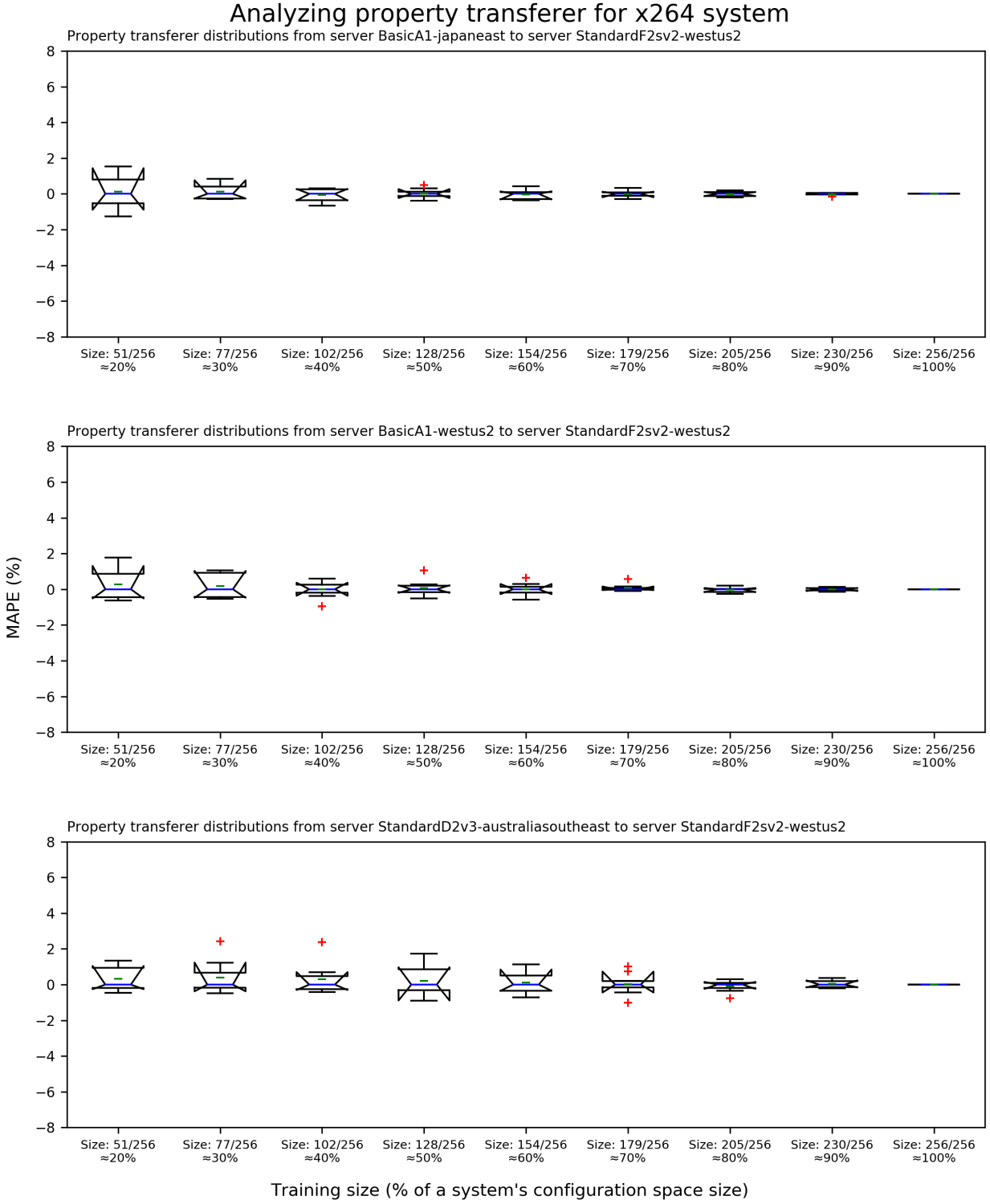
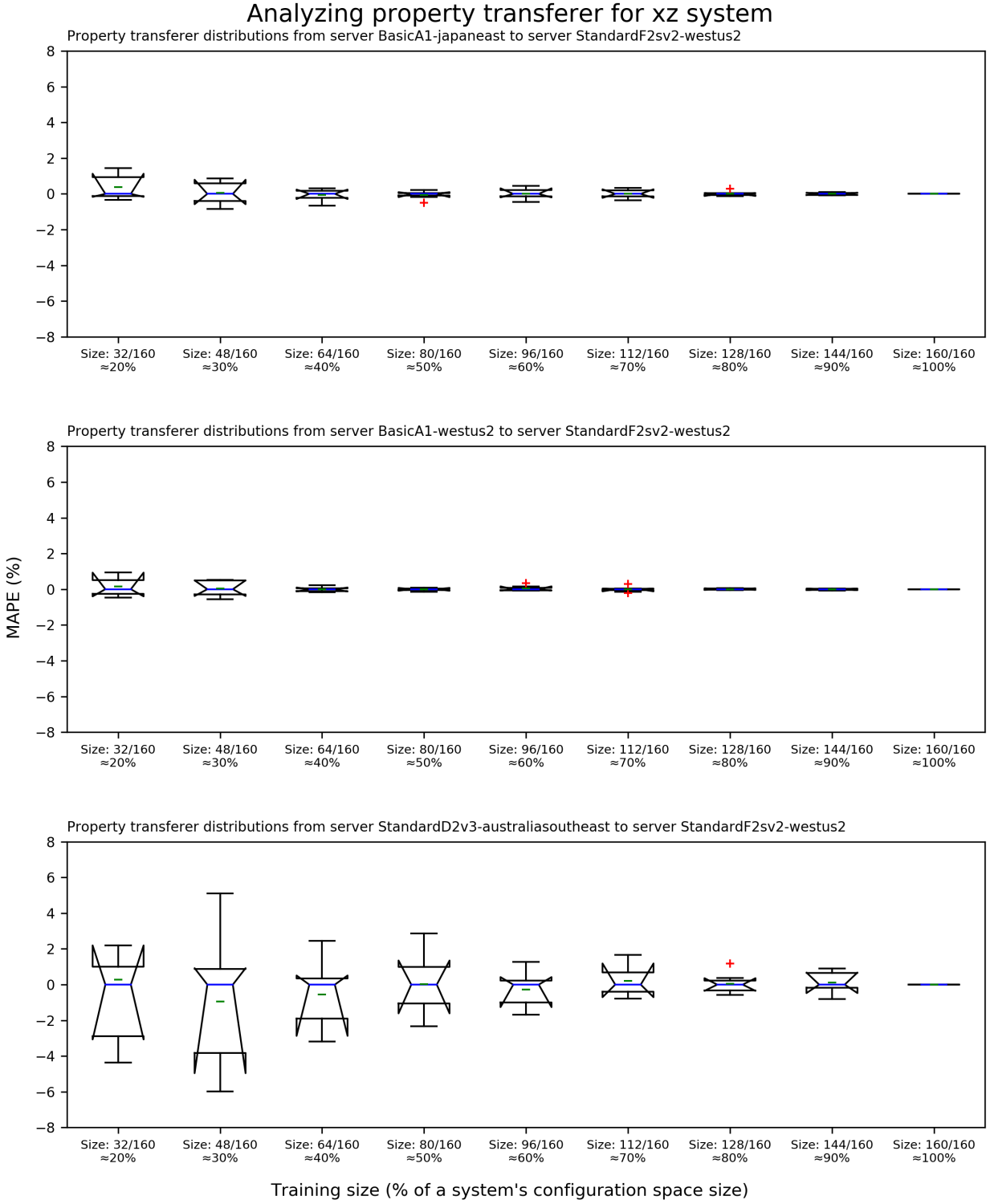


Figure 4.26: Distributions of MAPE error of *regression-trees-based* transferrers, when transferring the *compression time* metric for *xz* software on heterogeneous hardware, using different training sample sizes. Each subplot represents MAPE distributions for a particular hardware. Each box represents MAPE distribution for a particular training size. All boxes are normalized by median deduction. Each observation represents a MAPE value for a single experiment replication.



regression trees *RT* as our property transferring models (see Section 4.2.3 for details), we also performed a comprehensive evaluation of simple linear regression models *SLR*, since linear regression displayed high transferring performance in previous research (Jamshidi et al., 2017a; Valov et al., 2017). We assessed the quality of transferrers using *MAPE* and *LOOCV* validation.

Evaluation shows that performance of linear transferrers highly varies from one system to another. Figure 4.15 presents evaluation results by displaying distributions of *linear-based transferrers' MAPE* with increase of transferrers' training sample size. While for some systems, e.g. BZIP2 and GZIP, linear transferrers achieve *MAPE* smaller than 5%, for FLAC linear transferrers can barely achieve 20% for the majority of hardware platforms. Linear transferrers also exhibit poor performance for XZ parallel compression software.

Figure 4.21 presents evaluation results by displaying distributions of *tree-based transferrers'* averaged *MAPE* with increase of transferrers' training sample size. Regression trees provide significantly better results for transferring FLAC, x264, and XZ software systems, and comparable results for BZIP2. Although regression trees provide slightly worse results for GZIP than linear models when assessing them using *LOOCV* on a training sample, regression trees still outperform linear models when assessing actually transferred Pareto frontiers for GZIP systems.

Figures 4.16 – 4.20 and Figures 4.22 – 4.26 present *MAPE* distributions of linear-based and tree-based transferrers respectively, that are specifically trained for transferring compression time metric values. Each figure displays distributions for a specific software system, while each distribution consists of corresponding experimental replications. All of the distributions are normalized by median deduction. Symmetry of distributions and minimal amount of outliers allows to utilize the arithmetic mean for linear transferrers quality approximation. *MAPE* distributions also exhibit variability reduction with increase of a training sample size, meaning that the mean of experimental replications becomes even more reliable for assessing transferrers quality with increase of training data size.

Assessing Pareto Frontiers Accuracy

Before we can compare actual, approximated, and transferred Pareto frontiers, we have to select a way of assessing their prediction quality. We regard a Pareto frontier as a binary classifier that separates all system's configurations into optimal and non-optimal ones on a given hardware. Therefore, to assess the quality of a Pareto frontier we can use any of the classical statistical measures for binary classifiers.

Researchers use different statistical measures when assessing binary classifiers in their works. We decided to include several basic measures in order to provide a comprehensive and intuitive description of how well do approximated and transferred Pareto frontiers classify configurations. Table 4.4 presents all statistical measures used in our work in a form of a *confusion matrix*. We selected matrix representation since it shows all measures in a concise and structured way. To make statistical measures more intuitive, we indicate their preferred values using special symbols. ($\nearrow$) indicates that higher values of a respective measure are preferred to lower ones, while ($\swarrow$) indicates that lower values are preferred instead.

A core of a confusion matrix is formed by a contingency table, showing frequency distributions of optimal and non-optimal configurations. These are the most basic classification metrics, from

Table 4.4: Statistical measures for assessing a Pareto frontier, represented as a confusion matrix

	Condition Positive: $P = TP + FN$	Condition Negative: $N = FP + TN$		
Predicted Condition Positive: $PCP = TP + FP$	True Positive: $TP\uparrow$	False Positive: $FP\downarrow$	Positive Predictive Value: $PPV\uparrow = TP/PCP$	False Discovery Rate: $FDR\downarrow = FP/PCP$
Predicted Condition Negative: $PCN = FN + TN$	False Negative: $FN\downarrow$	True Negative: $TN\uparrow$	False Omission Rate: $FOR\downarrow = FN/PCN$	Negative Predictive Value: $NPV\uparrow = TN/PCN$
	True Positive Rate: $TPR\uparrow = TP/P$	False Positive Rate: $FPR\downarrow = FP/N$	F_1 score: $F_1\uparrow = 2 \times (PPV \times TPR)/(PPV + TPR)$	
	False Negative Rate: $FNR\downarrow = FN/P$	True Negative Rate: $TNR\uparrow = TN/N$	Matthews correlation coefficient: $MCC\uparrow = \sqrt{PPV \times NPV \times TPR \times TNR} - -\sqrt{FDR \times FOR \times FPR \times FNR}$	

which all other metrics can be derived. *True positive $TP\uparrow$ (negative $TN\uparrow$)* is an amount of actually optimal (non-optimal) configurations correctly classified as such by a Pareto frontier. *False positive $FP\downarrow$ (negative $FN\downarrow$)* is an amount of actually non-optimal (optimal) configurations misclassified as optimal (non-optimal) by a frontier.

The following measures allows a practitioner to answer specific questions about efficiency of a frontier in general. *True positive rate ($TPR\uparrow$)* shows a probability that a frontier contains all actually optimal configurations. *True negative rate ($TNR\uparrow$)* shows how likely will a frontier leave out all actually non-optimal configurations.

The next block of measures allows a practitioner to answer questions about classification results by a Pareto frontier. *Positive predictive value ($PPV\uparrow$)* represents a probability that a configuration, classified as optimal by a Pareto frontier, is truly optimal. *Negative predictive value ($NPV\uparrow$)* shows how likely a configuration, classified as non-optimal by a Pareto frontier, is truly non-optimal.

Although previously presented statistical measures provide information about a Pareto frontier quality, these measures are best regarded together in groups, since this way they provide a more comprehensive understanding of a frontier's performance. Therefore, we also included an 'integral' measure of a binary classification behavior. *Matthews correlation coefficient (MCC)* is considered to be one of the best measures for working with data that has strong quantitative differences between classes of observations. MCC takes values in $[-1, +1]$, where -1 corresponds to a completely misclassifying frontier, 0 to a random frontier, and $+1$ to a perfect frontier.

Approximated Frontiers Accuracy

To answer RQ3 we assess the accuracy of Pareto frontiers, approximated using our methodology. We calculate an approximated Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{src}}$ based on system properties $\mathbb{P}$ approximated for all configurations $\mathbb{C}$ using tree-based predictors that we train using samples of measured configurations $\mathbb{C}_{trn}$ (see Section 4.2.2). We regard approximated Pareto frontiers as binary classifiers

Figure 4.27: Distributions of statistical classification measures, characterizing Pareto frontiers *approximated using regression trees* on *BasicA1-japaneast* Azure server. Each subplot represents distributions for a particular software. Each line represents a particular measure: TPR (green), TNR (red), PPV (blue), NPV (orange), MCC (violet).

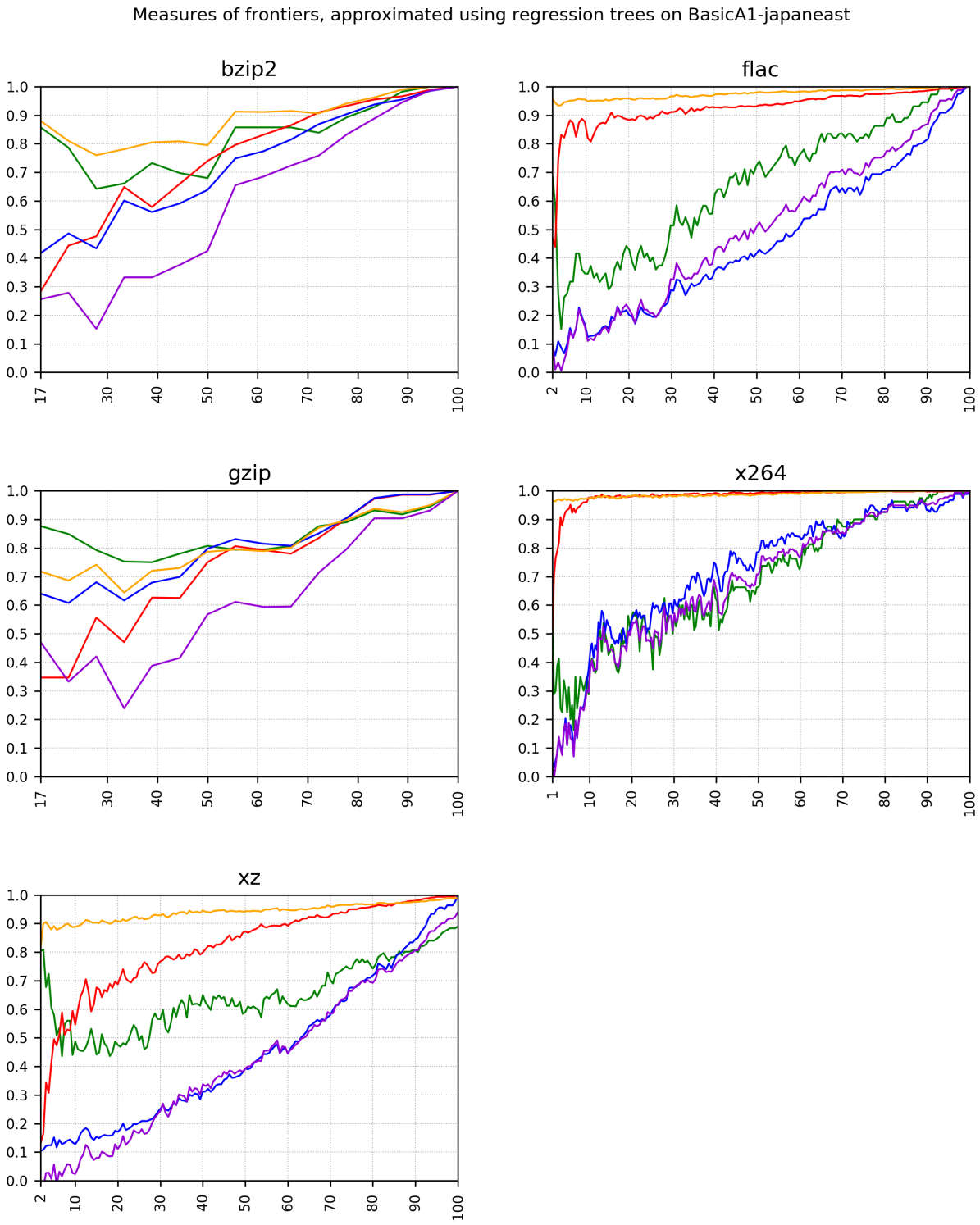


Figure 4.28: Distributions of MCC measure, characterizing Pareto frontiers *approximated using regression trees* on *BasicA1-japaneast* Azure server. Each subplot represents distributions for a particular software. Each box represents a distribution for a particular training sample size. Each observation represents an MCC value for a single experiment replication.

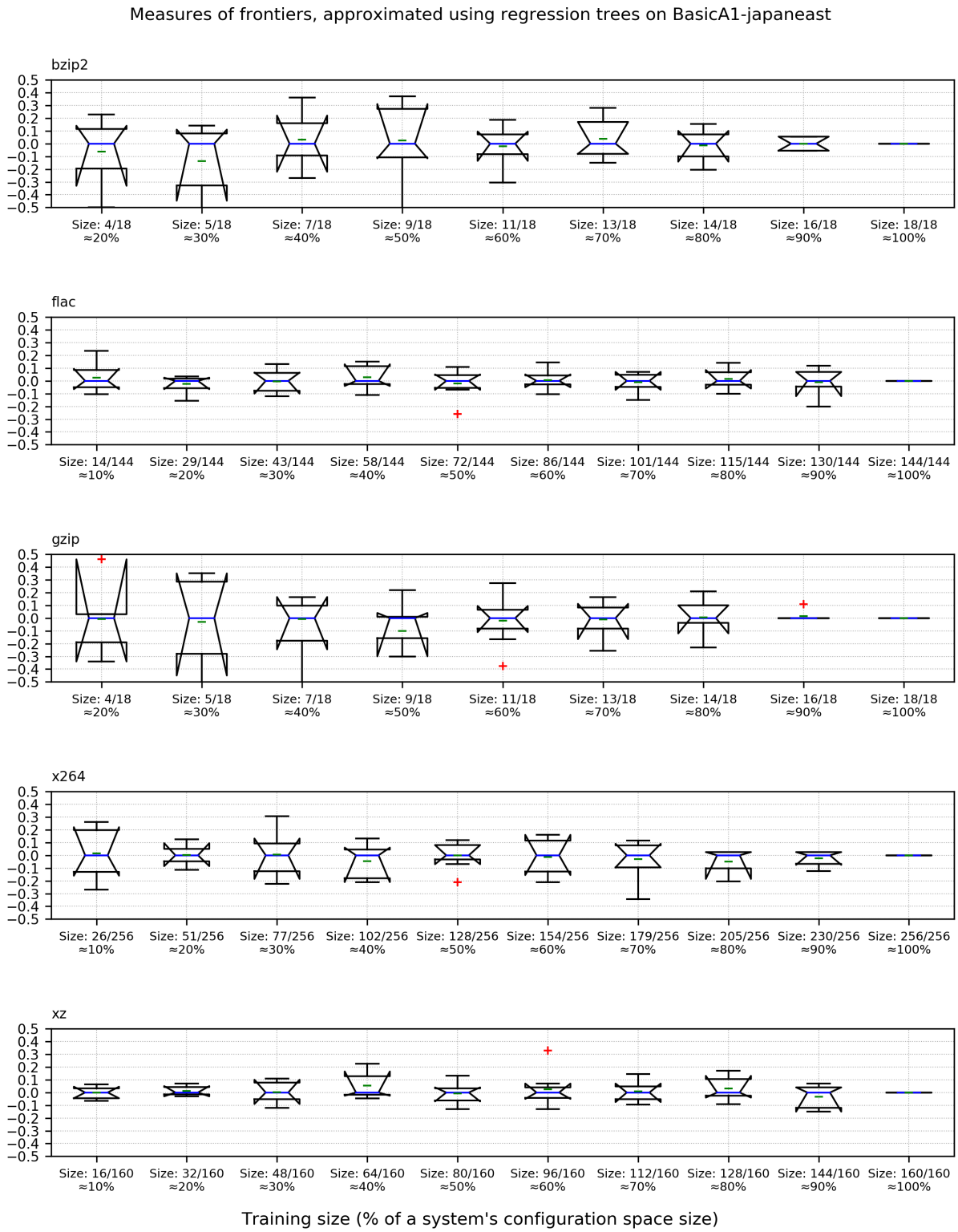


Figure 4.29: Distributions of statistical classification measures, characterizing Pareto frontiers *approximated using regression trees* on *StandardD2v3-australiasoutheast* Azure server. Each subplot represents distributions for a particular software. Each line represents a particular measure: TPR (green), TNR (red), PPV (blue), NPV (orange), MCC (violet).

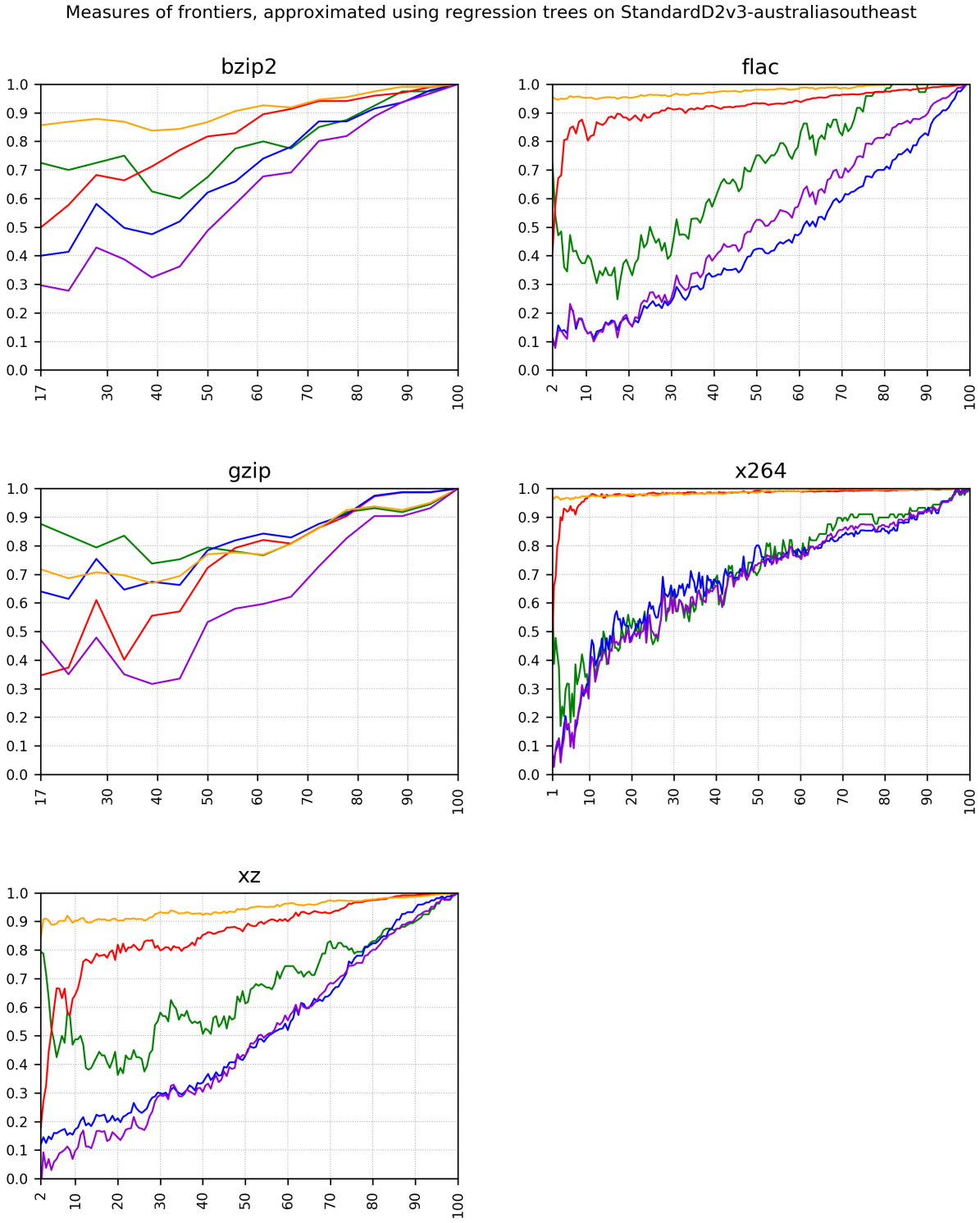
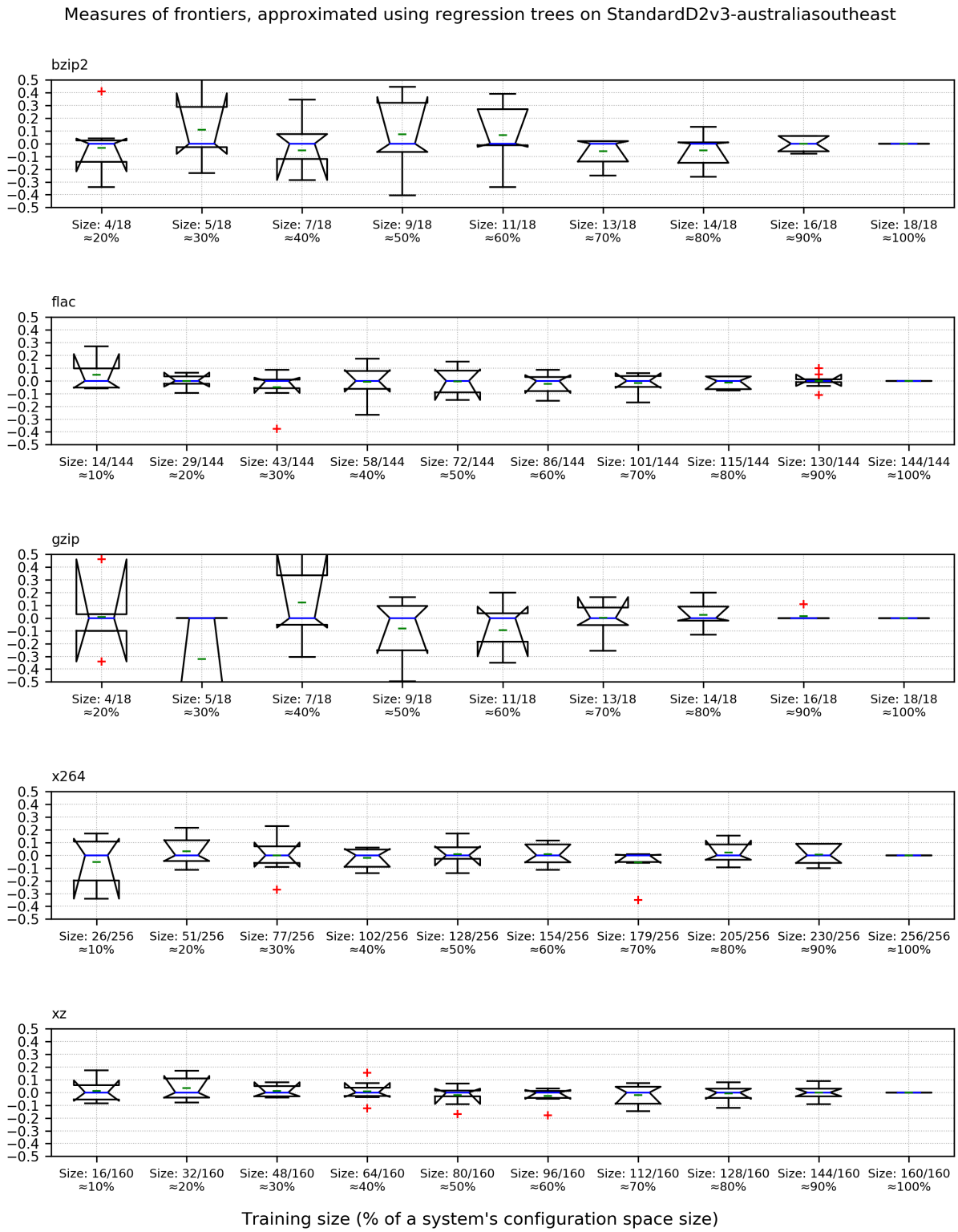


Figure 4.30: Distributions of MCC measure, characterizing Pareto frontiers *approximated using regression trees* on *StandardD2v3-australiasoutheast* Azure server. Each subplot represents distributions for a particular software. Each box represents a distribution for a particular training sample size. Each observation represents an MCC value for a single experiment replication.



and evaluate them using presented statistical measures (see Section 4.3.2). Figure 4.27 and Figure 4.29 show evaluation results of Pareto frontiers of all software systems for 'BasicA1-japaneast' and 'StandardD2v3-australiasoutheast' hardware environments respectively (see Table 4.1), approximated using tree-based predictors that are trained using samples $\mathbb{C}_{trn}$ of different sizes.

First of all, we observe that TNR and NPV demonstrate high values almost instantly, and provide almost perfect results for FLAC, x264, and XZ systems. This means that a frontier can very efficiently and with high certainty catch non-optimal configurations. However, this happens because classification categories are highly unbalanced in our case, since a number of optimal configurations in a system configuration space is generally much smaller than a number of non-optimal ones. Because of that, even if a frontier classifies all configurations as non-optimal, TNR and NPV might demonstrate high values in some cases. This effect becomes more apparent with a larger system's configuration space. Therefore, we cannot claim that approximated frontiers exhibit high classification accuracy based on TNR and NPV values only.

Secondly, we observe that TPR and PPV exhibit a gradual growth with increasing predictors' training sample sizes. This means that the ability of a frontier to capture optimal configurations and certainty that an optimally-classified configuration is truly optimal, both highly depend on the training sample size. We can explain this behavior by several major factors. First of all, classes of optimal and non-optimal configurations are highly unbalanced in our case. Since our approach cannot impose any restrictions on a sampling process and has to work with randomly-sampled data, truly optimal configurations on average become significantly underrepresented in the predictors' training samples. Thus, TPR and PPV gradually improve with the availability of new truly optimal configurations. Secondly, the inability to accurately capture optimal configurations by an approximated frontier based on small random samples of measured configurations lies in the structure of regression trees. Regression trees are limited by sampled property values and cannot output any value that is smaller than a minimal or larger than a maximal sampled value. Therefore, when regression trees are limited by a min-max interval of a random sample, they will be limited to a subinterval of possible values and consequently only to a part of an actual frontier, making accurate approximation of the whole frontier not possible.

Finally, we observe that MCC demonstrates almost linear growth approximately from 0 to 1, while avoiding negative values. This means that the presented Pareto frontier overall starts as a nearly-random classifier, but with the increase of predictors' training samples improves into an almost-perfect classifier, while avoiding complete misclassification of configurations. Figures 4.28 and 4.30 show distributions of the MCC measure across different experiment replications, characterizing approximated Pareto frontiers on *BasicA1-japaneast* and *StandardD2v3-australiasoutheast* hardware environments respectively. Symmetry of these distributions and minimal amount of outliers allow to use the arithmetic mean for approximation of MCC values.

To sum up, we evaluated our approach for approximating frontiers that works by individually predicting system's properties using general-purpose machine-learning models trained using random samples of measured configurations. We demonstrated that in general our approach works, but its overall quality is linearly dependent on the training sample's size. In order to improve accuracy of our approach in future work, we might need to relax some initial assumptions like the ability to control the sampling process.

Figure 4.31: Distributions of statistical classification measures, characterizing Pareto frontiers *transferred* from *BasicA1-japaneast* to *StandardG1-eastus2* Azure servers, using different transferring models. Each subplot represents distributions for a particular transferrer. Each line represents a particular measure: TPR (green), TNR (red), PPV (blue), NPV (orange), MCC (violet).

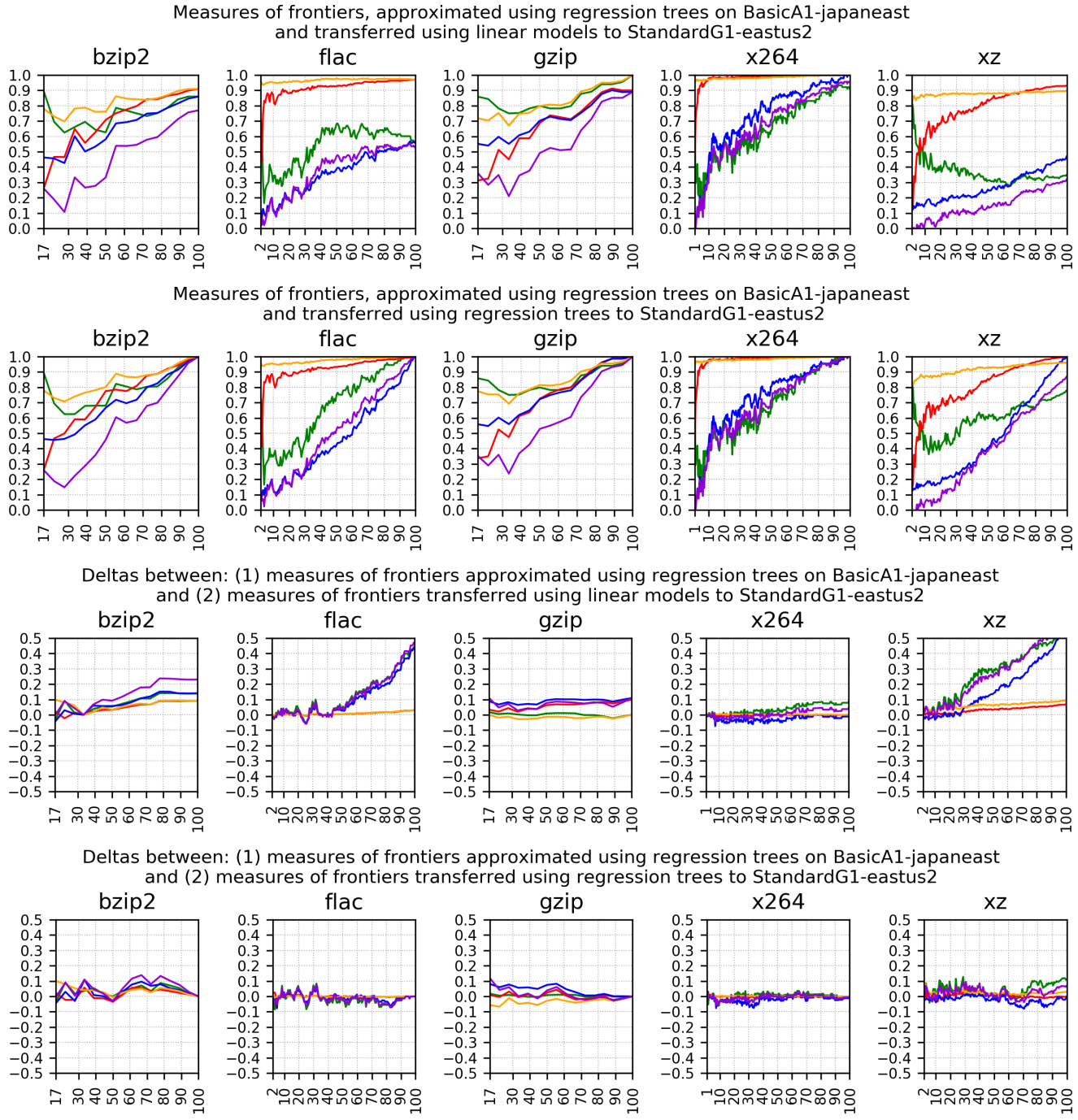


Figure 4.32: Distributions of statistical classification measures, characterizing Pareto frontiers *transferred* from *StandardD2v3-australiasoutheast* to *StandardF2sv2-westus2* Azure servers, using different transferring models. Each subplot represents distributions for a particular transferrer. Each line represents a particular measure: TPR (green), TNR (red), PPV (blue), NPV (orange), MCC (violet).

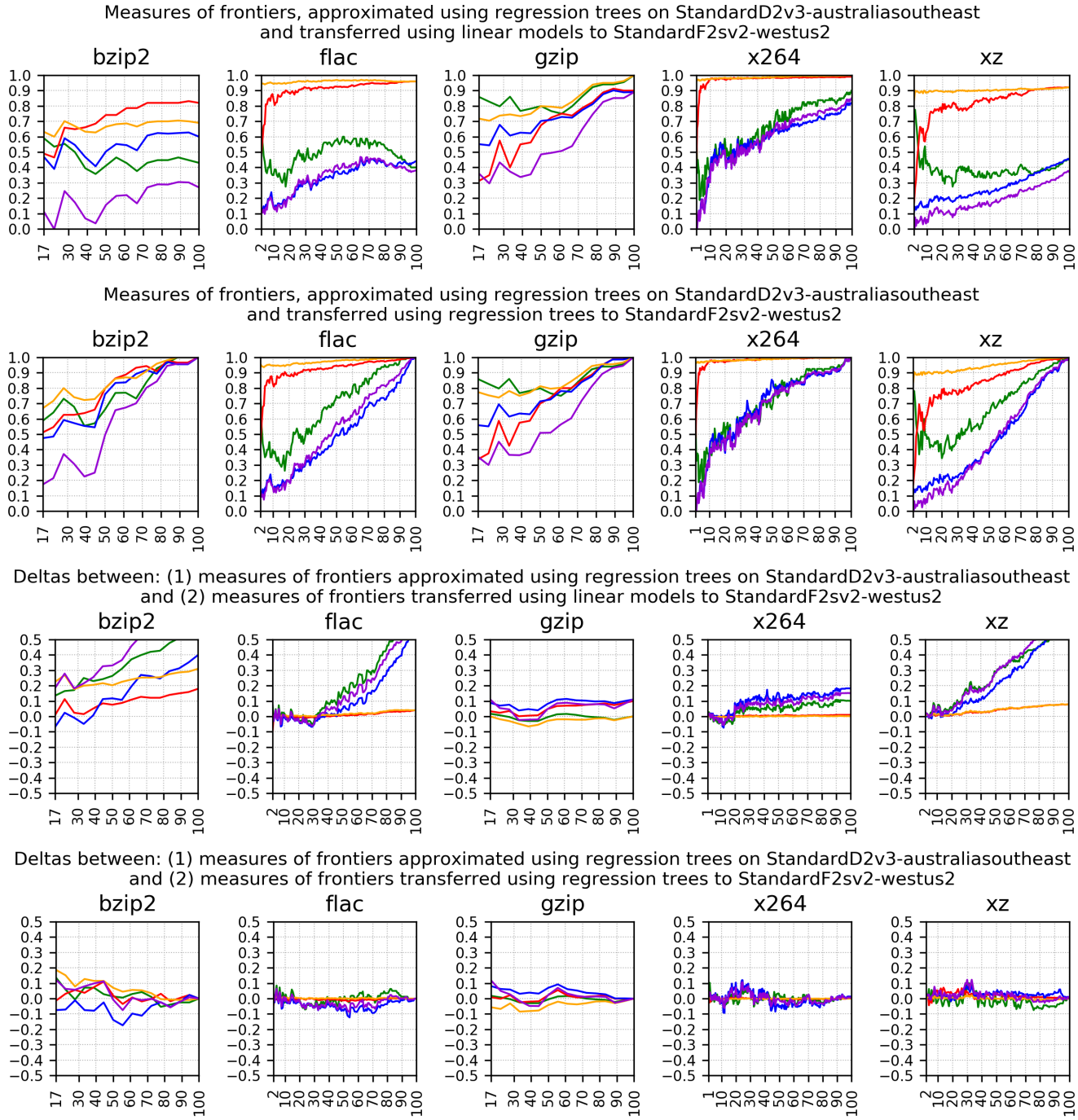


Figure 4.33: Distributions of MCC measure, characterizing Pareto frontiers *approximated using regression trees* on *StandardD2v3-australiasoutheast* Azure server. Each subplot represents distributions for a particular software. Each box represents a distribution for a particular training sample size. Each observation represents an MCC value for a single experiment replication.

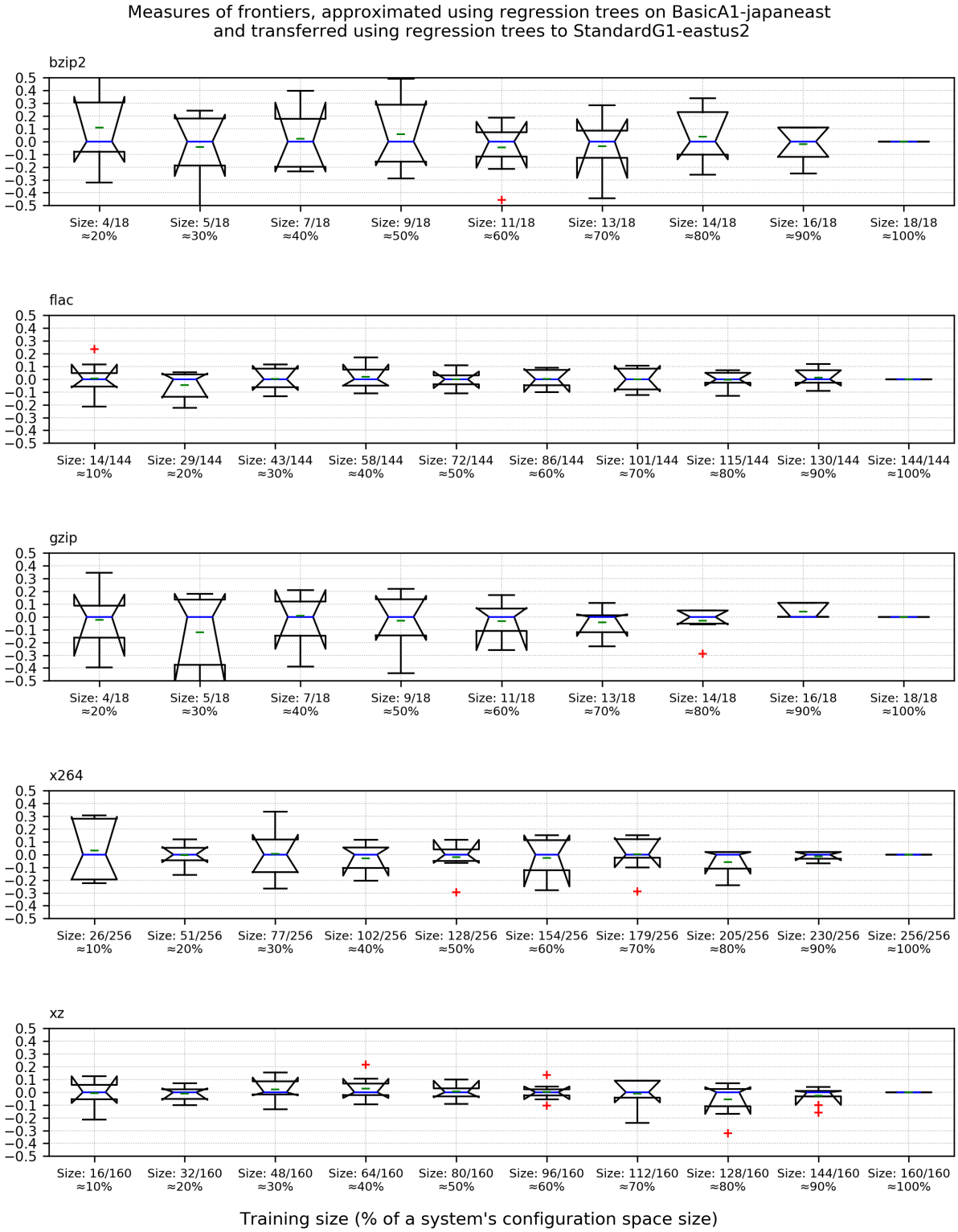
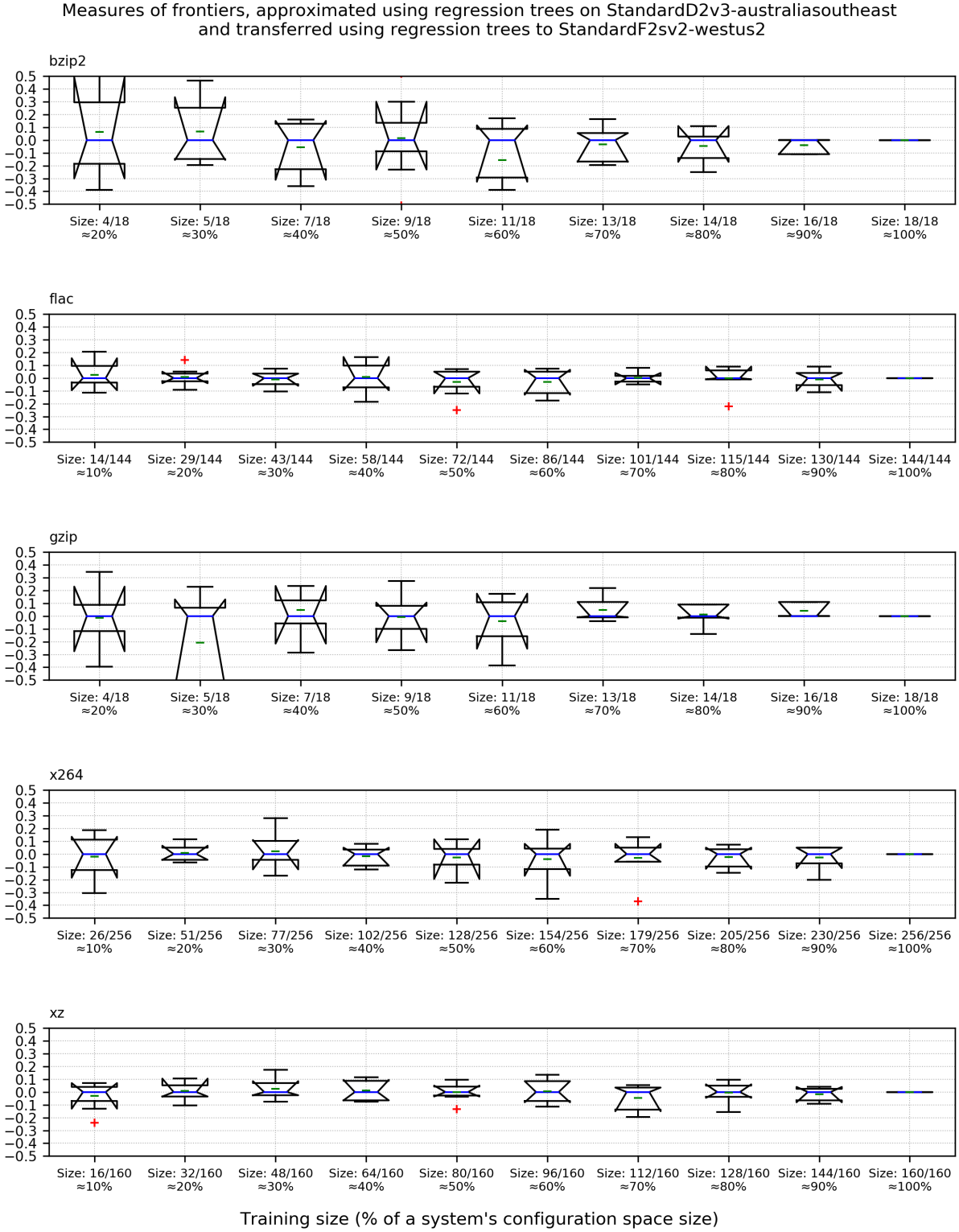


Figure 4.34: Distributions of statistical classification measures, characterizing Pareto frontiers *approximated using regression trees* on *BasicA1-japaneast* Azure server. Each subplot represents distributions for a particular software. Each line represents a particular measure: TPR (green), TNR (red), PPV (blue), NPV (orange), MCC (violet).



Transferred Frontiers Accuracy

To answer RQ4 we assess the accuracy of Pareto frontiers transferred using our approach. We acquire a transferred Pareto frontier $\widehat{\mathbb{C}}^{PF}_{h_{dst}}$ by transferring approximated values of all properties $\mathbb{P}$ using transferrers that we train using samples of configurations $\mathbb{C}_{both}$ measured on both source and destination hardware (see Section 4.2.4). Figures 4.31 and 4.32 present evaluation results of transferred Pareto frontiers of all studied software systems for linear-based and tree-based transferrers. Figure 4.31 presents frontiers transferred from 'BasicA1-japaneast' to 'StandardG1-eastus2' hardware environments, while Figure 4.32 presents frontiers transferred from 'StandardD2v3-australiasoutheast' to 'StandardF2sv2-westus2' hardware environments (see Table 4.1 for comparison).

Although linear regression demonstrated strong results for transferring prediction models across heterogeneous hardware previously (Jamshidi et al., 2017b; Valov et al., 2017), Figures 4.31 and 4.32 clearly demonstrate that regression trees completely outperform linear models for the majority of studied software systems. This difference is especially strong for FLAC and XZ software systems where MCC for linear-based transferring barely reaches 0.5 and 0.3 respectively.

To visually represent how much distortion to the approximated frontier the transferring process adds, we calculate the difference between the approximated and transferred frontiers for all statistical measures and present them on Figures 4.31 and 4.32. Thus, we can observe that the transferring process using regression trees has a very limited impact on distortion of an approximated frontier even for small training samples sizes, while transferring using linear models might significantly degrade the quality of a transferred Pareto frontier especially for large training sample sizes.

To sum up, we have demonstrated that transferring of approximated Pareto frontiers across heterogeneous hardware environments using unpruned regression trees is possible and doesn't significantly affect the resulting frontier's accuracy. However, a transferred frontier cannot demonstrate a high quality if an original approximated frontier doesn't show high classification results. Therefore, in future work we plan to improve our approach for minimalistic practical approximation of Pareto frontiers.

4.4 Threats to Validity

To increase internal validity of our research, during evaluation of our methodology, we trained properties' predictors and transferrers using random samples of measured configurations, and for each training size we generated 10 different random samples. Thus, all statistical measures that describe predictors, transferrers, and frontiers, are averaged over 10 different instances. Therefore, we avoid bias caused by random variations in models' training samples.

To increase external validity of our research, we performed an evaluation of our approach using five configurable software systems with different code bases, features, configuration space sizes, parallelization capabilities, and application domains. We benchmarked the selected software systems on 34 heterogeneous hardware platforms with different CPU models, available CPU cores, RAM sizes, and storage types. When benchmarking software systems, we measured each configuration

10 times and averaged over these measurements to get final properties' values. This allowed us to avoid bias induced by random aberrations during the benchmarking process.

We tried to address the most obvious threats to internal and external validity of our work, but we acknowledge that this might not be enough. Although we explored a variety of general-purpose software systems including parallelized ones, we suspect that there might be other configurable systems with different features, architectures, or application domains, whose properties might exhibit completely different unsystematic behavior across variable hardware. Moreover, we expect this behavior to occur when a practitioner redeploys a particular software system that is optimized for a specific hardware architecture. For example, when a system that supports GPU-acceleration is redeployed to a hardware without a dedicated graphical unit. We plan to investigate such software-hardware interaction in future work.

4.5 Summary

In the current chapter we proposed and evaluated a practical, easy-to-use, and black-box approach based on general-purpose machine-learning models for approximation and transferring of Pareto frontiers of optimal configurations. We perform approximation of a frontier by (1) building an unpruned regression tree model for each property to act as a predictor, and then (2) combining properties' predictions into an approximated frontier. Our evaluation shows a strong decreasing trend in predictors' error and a linearly increasing trend of an overall classification accuracy of a resulting approximated frontier, with increase of predictors' training sample sizes. We perform transferring of a frontier by (1) building an unpruned regression tree model for each property to act as a transferrer, and then (2) combining transferred values of the approximated frontier. Our evaluation shows a strong decreasing trend in transferrers' error and a linearly increasing trend of a transferred frontier accuracy, with increase of transferrers' training sample sizes. Moreover, the overall accuracy of a transferred Pareto frontier mostly depends on the approximated Pareto frontier's accuracy than on the transferring process itself.

Chapter 5

Conclusion and Future Work

In this work, we proposed and evaluated approaches for transferring property prediction of configurable software systems across a collection of heterogeneous hardware environments. Both of these approaches are practical and minimalistic. They regard software systems as black-boxes and train property prediction and transferring models using small random samples of measured configurations.

The first method addresses the problem of configurable system performance prediction transfer across a heterogeneous hardware cluster. This method (1) builds a performance prediction model on source hardware and (2) transforms predictions of this model to a destination hardware by using a separate transferring model. Analysis of the method demonstrated that cross-platform similarity of performance distributions directly correlates with the structure of performance prediction models and overall quality of performance prediction and transferring. Experiments on three different configurable software systems demonstrate that it is possible to build a reliable linear transferrer using a small sample of configurations $\mathbb{C}_{both} : size(\mathbb{C}_{both}) \in [5, 10]$ measured on both source and destination hardware environments. Analysis of different sampling techniques demonstrated that the best way to generate $\mathbb{C}_{both}$ is by resampling from configurations measured on source hardware environment using a quasi-random technique (Sobol sampling). Our experiments show strong decreasing trends of MAPE for both predictors and transferrers with increase of a training sample size, while achieving high accuracy (less than 10% MAPE) for the majority of transfers.

The second method addresses the problem of transferring prediction about relative configuration optimality of a software system across a heterogeneous hardware cluster, by extrapolating the first method on multiple system properties. This method (1) trains a prediction model (e.g. a regression tree) for each analyzed software system property (e.g. compression time, compressed size) by using configurations measured on the source hardware environment, (2) approximates properties of all unmeasured configurations using trained predictors and builds an approximated Pareto frontier on the source hardware, (3) trains a transferrer model (e.g. a linear model or a regression tree) for each analyzed system property, by using configurations measured on both source and destination hardware environments, and (4) transfers the approximated Pareto frontier from source to destination hardware, by individually transferring approximated properties using the trained transferrers, thus acquiring a transferred Pareto frontier on destination hardware.

We evaluate the proposed methodology by regarding approximated and transferred Pareto frontiers as binary classifiers, and assessing them using classical statistical measures like Matthews

Correlation Coefficient (MCC). Experiments on five different configurable software systems demonstrate a strong linearly increasing trend in average MCC values and a strong decreasing trend in MCC's variability with increase of the training sample size for both approximated and transferred frontiers. Experiments demonstrate that the classification accuracy of a transferred frontier mostly depends on the approximation process, while the transferring process only adds a minor error to the classification even for small training sample sizes.

Nevertheless, our research has some limitations. First of all, the studied configurable software systems are relatively small in terms of the number of features and the number of valid system configurations. This might simplify the problem of Pareto frontier approximation and transferring. To check this assumption in the future work we plan to analyze much more complex configurable software, such as, FreeBSD and Linux kernels (Berger et al., 2010).

Secondly, we analyzed only two differing properties for all studied configurable software systems. A larger number of analyzed properties might reduce the quality of the Pareto frontier approximation and transferring process. In the future work we plan to analyze our approach for approximation and transferring of frontiers that are based on three or more configuration properties.

Thirdly, when building property prediction and transferring models, we only take into account software features and do not analyze any hardware features. Analysis of hardware features might be beneficial when transferring a frontier across a heterogeneous hardware cluster, because this information might help to even further save benchmarking effort by more efficient transferring of property values across hardware with similar configurations. Moreover, we assume that all hardware platforms in the cluster use the same operating system and compiler, which might not be the case in a real-world scenario. Therefore, in the future work we plan not only to use hardware features, but also perform transferring across different combinations of hardware platforms and system software.

Fourthly, when benchmarking the studied software systems across heterogeneous hardware, we measured a single software on a single hardware with a fixed workload. However, in a real-world scenario multiple software systems might be deployed on a single platform and these systems might experience different workloads (Delimitrou and Kozyrakis, 2014). Thus, in the future work we plan to experiment with deploying multiple software systems on a single platform and experimenting with varying software workloads.

Fifthly, we used pseudo-random sampling when exploring the configuration spaces of the studied software in order to mimic the worst case scenario of impossibility to make any sampling choices by the end user of our approach. We selected pseudo-random sampling as a simple method, which may not provide a high quality configurations diversity. We understand that in a real-world scenario a user might have an extremely biased sample that cannot be simulated by the pseudo-random sampling, however, the analysis of this scenario was out of scope of the current research. Nevertheless, in the future work we plan to further explore different sampling techniques in order to provide a better assessment of the worst-case behavior of our methodology.

Finally, in our approach we focused on approximation and transferring of exact Pareto frontiers. However, a user might be interested not only in Pareto optimal configurations, but in near-optimal configurations as well. Inclusion of near-optimal configuration would allow to use additional methods for approximation and transferring of Pareto frontiers. Therefore, in the future work we plan to analyze and compare different methods for approximate Pareto frontier generation (Olaechea et al., 2014).

References

Adil Ali Abdelaziz, Wan Mohd Nasir Wan Kadir, and Addin Osman. Comparative analysis of software performance prediction approaches in context of component-based system. *International Journal of Computer Applications*, 23(3):15–22, June 2011. doi: 10.5120/2870-3725.

N. R. Adiga, G. Almasi, G. S. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. M. Cipolla, P. Crumley, K. M. Desai, A. Deutsch, T. Domany, M. B. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. E. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. A. Haring, D. Heidel, P. Heidelberger, L. M. Herger, D. Hoenicke, R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, E. Krevat, M. P. Kurhekar, A. P. Lanzetta, D. Lieber, L. K. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. E. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. B. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. K. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. D. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. B. Tremaine, M. Tsao, A. R. Umamaheshwaran, P. Verma, P. Vranas, T. J. C. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. T. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. K. Seager, J. S. Vetter, and K. Yates. An overview of the bluegene/l supercomputer. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 60–60. IEEE, Nov 2002. doi: 10.1109/SC.2002.10017. URL https://ieeexplore.ieee.org/abstract/document/1592896.

David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, January 1991. ISSN 0885-6125, 1573-0565. doi: 10.1023/A:1022689900470. URL https://doi.org/10.1023/A:1022689900470.

J. Antony. *Design of Experiments for Engineers and Scientists*. Butterworth-Heinemann, 2003.

Domagoj Babić and Frank Hutter. Spear theorem prover. In *SAT'08: Proceedings of the SAT 2008 Race*, 2008. URL http://www.domagoj-babic.com/index.php/Pubs/SAT08.

Simonetta Balsamo and Moreno Marzolla. Performance evaluation of uml software architectures with multiclass queueing network models. In *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, pages 37–42, New York, NY, USA, 2005. ACM. ISBN 1-59593-087-6. doi: 10.1145/1071021.1071025. URL http://doi.acm.org/10.1145/1071021.1071025.

Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. Feature-to-code mapping in two large product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, pages 498–499, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15579-6. doi: 10.1007/978-3-642-15579-6_48. URL https://dx.doi.org/10.1007/978-3-642-15579-6_48.

Michel Berkelaar, Kjell Eikland, Peter Notebaert, et al. lpsolve: Open source (mixed-integer) linear programming system. *Eindhoven U. of Technology*, 2004.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738. URL https://dl.acm.org/citation.cfm?id=1162264.

Léon Bottou, Olivier Chapelle, Dennis DeCoste, and Jason Weston. *Large-Scale Kernel Machines*. Neural Information Processing. The MIT Press, Cambridge, MA, USA, 2007. ISBN 0262026252.

Ryan L. Braby, Jim E. Garlick, and Robin J. Goldstone. Achieving order through chaos: the llnl hpc linux cluster experience. In *The 4th International Conference on Linux Clusters: The HPC Revolution*, June 2003. URL https://e-reports-ext.llnl.gov/pdf/243159.pdf.

Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, August 1996. ISSN 1573-0565. doi: 10.1007/BF00058655. URL https://link.springer.com/article/10.1007%2FBF00058655.

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, October 2001. ISSN 0885-6125. doi: 10.1023/A:1010933404324. URL https://dl.acm.org/citation.cfm?id=570182.

Leo Breiman. *Classification and Regression Trees*. Routledge, New York, NY, USA, 2017. URL https://doi.org/10.1201/9781315139470.

Eric Allen Brewer. *Portable High-performance Supercomputing: High-level Platform-dependent Optimization*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1994. URL https://dl.acm.org/citation.cfm?id=921360.

Eric Allen Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 80–91, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. doi: 10.1145/209936.209946. URL http://doi.acm.org/10.1145/209936.209946.

Peter N. Brown, Robert D. Falgout, and Jim E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000. doi: 10.1137/S1064827598339141. URL https://doi.org/10.1137/S1064827598339141.

John G. Cleary and Leonard E. Trigg. K*: An instance-based learner using an entropic distance measure. In Armand Prieditis and Stuart Russell, editors, *Machine Learning Proceedings 1995*, pages 108–114. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995. ISBN 978-1-55860-377-6. doi: 10.1016/B978-1-55860-377-6.50022-0. URL http://www.sciencedirect.com/science/article/pii/B9781558603776500220.

William W. Cohen. Fast effective rule induction. In Armand Prieditis and Stuart Russell, editors, *Proceedings of the 12th International Conference on Machine Learning*, ICML'95, pages 115–123, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-377-8, 978-1-55860-377-6. URL http://dl.acm.org/citation.cfm?id=3091622.3091637.

L. Collin. XZ Utils. http://tukaani.org/xz/, 2020. Accessed April. 17th, 2016.

Collin, L. XZ Utils Manual. https://linux.die.net/man/1/xz, 2020. Accessed April. 17th, 2016.

Vittorio Cortellessa and Raffaela Mirandola. PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Science of Computer Programming*, July 2002.

Marc Courtois and Murray Woodside. Using regression splines for software performance analysis. In *Proceedings of the 2nd International Workshop on Software and Performance*, WOSP '00, pages 105–114, New York, NY, USA, 2000. ACM. ISBN 1-58113-195-X. doi: 10.1145/350391.350416. URL http://doi.acm.org/10.1145/350391.350416.

Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, NY, USA, 1991. ISBN 0-471-06259-6. URL https://dl.acm.org/citation.cfm?id=129837.

IBM ILOG CPLEX. V12. 1: Users manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

C. Croarkin and P. Tobias. NIST/SEMATECH e-Handbook of Statistical Methods. http://www.itl.nist.gov/div898/handbook/, 2013.

Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 162–171, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0102-2. doi: 10.1145/1995896.1995923. URL http://doi.acm.org/10.1145/1995896.1995923.

Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127144, February 2014. ISSN 0362-1340. doi: 10.1145/2644865.2541941. URL https://doi.org/10.1145/2644865.2541941.

Sylvain Durrleman and Richard Simon. Flexible regression models with cubic splines. *Statistics in Medicine*, 8(5):551–562, 1989. URL https://onlinelibrary.wiley.com/doi/10.1002/sim.4780080504.

Rüdiger Esser and Renate Knecht. Intel paragon XP/S - architecture and software environment. In Hans-Werner Meuer, editor, *Supercomputer '93*, pages 121–141, Berlin, Heidelberg, 1993. Springer-Verlag. ISBN 978-3-642-78348-7. doi: 10.1007/978-3-642-78348-7_13. URL https://link.springer.com/chapter/10.1007%2F978-3-642-78348-7_13.

Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 124–133, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-612-2. URL http://dl.acm.org/citation.cfm?id=645528.657623.

Jerome H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1): 1–67, March 1991. ISSN 0090-5364; 2168-8966/e. doi: 10.1214/aos/1176347963. URL https://projecteuclid.org/euclid.aos/1176347963.

Gailly, J.-l. and Adler, M. GNU Gzip, free, open source, and patent free data compressor. https://www.gnu.org/software/gzip/, 2016a.

Gailly, J.-l. and Adler, M. GNU Gzip Manual. https://www.gnu.org/software/gzip/manual/gzip.html#Invoking-gzip, 2016b.

Jim E. Garlick and Chris M. Dunlap. Building chaos: An operating system for livermore linux clusters. Technical report, Lawrence Livermore National Lab., CA (US), February 2003.

Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 301–311, Piscataway, NJ, USA, November 2013. IEEE Press. ISBN 978-1-4799-0215-6. doi: 10.1109/ASE.2013.6693089. URL https://doi.org/10.1109/ASE.2013.6693089.

Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. Data-efficient performance learning for configurable systems. *Empirical Software Engineering*, 23(3):1826–1867, June 2018. ISSN 1573-7616. doi: 10.1007/s10664-017-9573-6. URL https://doi.org/10.1007/s10664-017-9573-6.

Eric A. Hansen and Shlomo Zilberstein. Monitoring the progress of anytime problem-solving. In *Proceedings of the 13th National Conference on Artificial Intelligence - Volume 2*, AAAI'96, pages 1229–1234, Portland, Oregon, 1996. AAAI Press. ISBN 0-262-51091-X. URL http://dl.acm.org/citation.cfm?id=1864519.1864569.

Nikolaus Hansen and Stefan Kern. Evaluating the cma evolution strategy on multimodal test functions. In Xin Yao, Edmund K. Burke, José A. Lozano, Jim Smith, Juan Julián Merelo-Guervós, John A. Bullinaria, Jonathan E. Rowe, Peter Tiňo, Ata Kabán, and Hans-Paul Schwefel, editors, *International Conference on Parallel Problem Solving from Nature - PPSN VIII*, pages 282–291, Berlin, Heidelberg, 2004. Springer. ISBN 978-3-540-30217-9. URL https://link.springer.com/chapter/10.1007/978-3-540-30217-9_29.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1994. ISBN 0023527617. URL https://dl.acm.org/citation.cfm?id=541500.

W. Daniel Hillis and Lewis W. Tucker. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, 36(11):31–40, November 1993. ISSN 0001-0782. doi: 10.1145/163359.163361. URL http://doi.acm.org/10.1145/163359.163361.

D. R. Hipp. SQLite. https://www.sqlite.org/, 2020. Accessed April. 15th, 2016.

Arthur E. Hoerl and Robert W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970. doi: 10.1080/00401706.1970.10488634. URL https://www.tandfonline.com/doi/abs/10.1080/00401706.1970.10488634.

Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 114–122, New York, NY, USA, 2006. ACM. ISBN 1-59593-264-X. doi: 10.1145/1152154.1152174. URL http://doi.acm.org/10.1145/1152154.1152174.

Adele E. Howe, Eric Dahlman, Christopher Hansen, Michael Scheetz, and Anneliese von Mayrhauser. Exploiting competitive planner performance. In Susanne Biundo and Maria Fox, editors, *Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning*, ECP '99, pages 62–72, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-67866-2, 978-3-540-44657-6. URL http://dl.acm.org/citation.cfm?id=647868.737100.

Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In Pascal Van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 233–248, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-44120-4, 978-3-540-46135-7. URL http://dl.acm.org/citation.cfm?id=647489.727303.

Frank Hutter, Youssef Hamadi, Holger H. Hoos, and Kevin Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In Frédéric Benhamou, editor, *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, CP'06, pages 213–228, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-46267-8, 978-3-540-46267-5. doi: 10.1007/11889205_17. URL http://dx.doi.org/10.1007/11889205_17.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Kevin P. Murphy. An experimental investigation of model-based parameter optimisation: SPO and beyond. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 271–278, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-325-9. doi: 10.1145/1569901.1569940. URL http://doi.acm.org/10.1145/1569901.1569940.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Automated configuration of mixed integer programming solvers. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR'10, pages 186–202, Berlin, Heidelberg, 2010a. Springer-Verlag. ISBN 3-642-13519-6, 978-3-642-13519-4, 978-3-642-13520-0. doi: 10.1007/978-3-642-13520-0_23. URL http://dx.doi.org/10.1007/978-3-642-13520-0_23.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Kevin Murphy. Time-bounded sequential parameter optimization. In Christian Blum and Roberto Battiti, editors, *Proceedings of the 4th International Conference on Learning and Intelligent Optimization*, LION'10, pages 281–298, Berlin, Heidelberg, 2010b. Springer-Verlag. ISBN 3-642-13799-7, 978-3-642-13799-0, 978-3-642-13800-3. URL http://dl.acm.org/citation.cfm?id=1893659.1893694.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin,

Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25565-6. doi: 10.1007/978-3-642-25566-3_40. URL http://dx.doi.org/10.1007/978-3-642-25566-3_40.

Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, January 2014. ISSN 0004-3702. doi: 10.1016/j.artint.2013.10.003. URL http://dx.doi.org/10.1016/j.artint.2013.10.003.

Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation (extended abstract). In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, IJCAI'15, Palo Alto, California, USA, July 2015. AAAI Press. ISBN 978-1-57735-738-4. URL https://www.aaai.org/ocs/index.php/IJCAI/IJCAI15/paper/view/10705.

Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 497–508, Piscataway, NJ, USA, 2017a. IEEE Press. ISBN 978-1-5386-2684-9. URL http://dl.acm.org/citation.cfm?id=3155562.3155625.

Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. Transfer learning for improving model predictions in highly configurable software. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '17, pages 31–41, Piscataway, NJ, USA, 2017b. IEEE Press. ISBN 978-1-5386-1550-8. doi: 10.1109/SEAMS.2017.11. URL https://doi.org/10.1109/SEAMS.2017.11.

Josh Coalson, Xiph.Org Foundation. FLAC – Free Lossless Audio Codec. https://xiph.org/flac/, 2019a.

Josh Coalson, Xiph.Org Foundation. FLAC Manual. https://xiph.org/flac/documentation_tools_flac.html, 2019b.

Ron Kohavi. The power of decision tables. In Nada Lavrac and Stefan Wrobel, editors, *Proceedings of the 8th European Conference on Machine Learning*, ECML'95, pages 174–189, Berlin, Heidelberg, 1995. Springer-Verlag. ISBN 3-540-59286-5, 978-3-540-59286-0. doi: 10.1007/3-540-59286-5_57. URL https://doi.org/10.1007/3-540-59286-5_57.

Matthias Kowal, Max Tschaikowski, Mirco Tribastone, and Ina Schaefer. Scaling Size and Parameter Spaces in Variability-Aware Software Performance Models (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015.

Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634 – 658, 2010. ISSN 0166-5316. doi: https://doi.org/10.1016/j.peva.2009.07.007. URL http://www.sciencedirect.com/science/article/pii/S016653160900100X. Special Issue on Software and Performance.

John Kubiatowicz and Anant Agarwal. Anatomy of a message in the Alewife multiprocessor. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 193–204, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2840-1. doi: 10.1145/2591635.2667168. URL http://doi.acm.org/10.1145/2591635.2667168.

Niels Landwehr, Mark Hall, and Eibe Frank. Logistic model trees. *Machine Learning*, 59(1-2): 161–205, May 2005. ISSN 0885-6125, 1573-0565. doi: 10.1007/s10994-005-0466-3. URL http://dx.doi.org/10.1007/s10994-005-0466-3.

Saskia Le Cessie and Hans C. Van Houwelingen. Ridge estimators in logistic regression. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 41(1):191–201, 1992. ISSN 00359254, 14679876. URL http://www.jstor.org/stable/2347628.

Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. doi: 10.1145/1229428.1229479. URL http://doi.acm.org/10.1145/1229428.1229479.

Levy, C. Sintel Trailer. https://media.xiph.org/, 2010.

Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In Pascal Van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 556–572, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-46135-7. URL http://dl.acm.org/citation.cfm?id=647489.727158.

David J.C. MacKay. Introduction to gaussian processes. *NATO Advanced Science Institutes Series F: Computer and Systems Sciences*, 168:133–166, January 1998.

Mahoney, M. Large Text Compression Benchmark. http://mattmahoney.net/dc/text.html, 2019.

Brent Martin. Instance-based learning: nearest neighbour with generalisation. Master's thesis, University of Waikato, Department of Computer Science, 1995. URL https://researchcommons.waikato.ac.nz/handle/10289/1095.

D.C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2008.

Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering*, 25(2):247–277, June 2018. ISSN 1573-7535, 0928-8910. doi: 10.1007/s10515-017-0225-2. URL https://doi.org/10.1007/s10515-017-0225-2.

Nine Inch Nails. Ghosts I-IV. https://archive.org/details/nineinchnails_ghosts_I_IV, 2008.

Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding random sat: Beyond the clauses-to-variables ratio. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, CP'04, pages 438–452, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30201-8. doi: 10.1007/978-3-540-30201-8_33. URL https://doi.org/10.1007/978-3-540-30201-8_33.

Rafael Olaechea, Derek Rayside, Jianmei Guo, and Krzysztof Czarnecki. Comparison of exact and approximate multi-objective optimization for software product lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, page 92101, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327404. doi: 10.1145/2648511.2648521. URL https://doi.org/10.1145/2648511.2648521.

Gurobi Optimization. Inc.,gurobi optimizer reference manual, 2015. *URL: http://www.gurobi.com*, 2014.

J. C. Petkovich, A. Oliveira, Y. Zhang, T. Reidemeister, and S. Fischmeister. DataMill: A Distributed Heterogeneous Infrastructure For Robust Experimentation. *Software: Practice and Experience*, pages n/a–n/a, 2015. ISSN 1097-024X. doi: 10.1002/spe.2382. URL http://dx.doi.org/10.1002/spe.2382.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. Cambridge Univ. Press, 1992.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-238-0. URL https://dl.acm.org/citation.cfm?id=152181.

Joaquin Quionero-candela, Carl Edward Rasmussen, and Christopher K.I. Williams. Approximation methods for gaussian process regression. In *Large-Scale Kernel Machines*, Neural Information Processing, pages 203–223, Cambridge, MA, USA, 2007. The MIT Press.

Carl Edward Rasmussen. *Gaussian Processes in Machine Learning*, pages 63–71. Springer-Verlag, Berlin, Heidelberg, 2004. ISBN 978-3-540-28650-9. doi: 10.1007/978-3-540-28650-9_4. URL https://doi.org/10.1007/978-3-540-28650-9_4.

Brian D Ripley. *Stochastic simulation*, volume 316. John Wiley & Sons, 2009.

Mark Roberts, Adele Howe, and Landon Flom. Learned models of performance for many planners. In *ICAPS 2007 Workshop AI Planning and Learning*, pages 36–40, 2007. URL http://icaps07-satellite.icaps-conference.org/workshop1/paper13.pdf.

Sylvain Roy. Nearest neighbor with generalization. *Christchurch, New Zealand*, 2002.

Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 342–352, Washington, DC, USA, November 2015. IEEE Computer Society. ISBN 978-1-5090-0025-8. doi: 10.1109/ASE.2015.45. URL https://doi.org/10.1109/ASE.2015.45.

Seward, J. and Mena F. Bzip2, free, open source, and patent free data compressor. https://sourceware.org/bzip2/, 2019a.

Seward, J. and Mena F. Bzip2 Manual. https://sourceware.org/bzip2/manual/manual.html#options, 2019b.

Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 167–177, Piscataway, NJ, USA, 2012a. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337243.

Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 167–177, Piscataway, NJ, USA, 2012b. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337243.

Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3):487–517, Sep 2012c. ISSN 1573-1367. doi: 10.1007/s11219-011-9152-9. URL https://doi.org/10.1007/s11219-011-9152-9.

Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 284–294, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786845. URL http://doi.acm.org/10.1145/2786805.2786845.

Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, August 2004. ISSN 0960-3174. doi: 10.1023/B:STCO.0000035301.49549.88. URL https://dl.acm.org/citation.cfm?id=1011939.

I.M Sobol and Yu.L Levitan. A pseudo-random number generator for personal computers. *Computers and Mathematics with Applications*, 37(4–5):33–40, 1999.

Marc Sumner, Eibe Frank, and Mark Hall. Speeding up logistic model tree induction. In Alípio Mário Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho, and João Gama, editors, *Proceedings of the 9th European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, ECMLPKDD'05, pages 675–683, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-29244-6, 978-3-540-29244-9, 978-3-540-31665-7. doi: 10.1007/11564126_72. URL https://doi.org/10.1007/11564126_72.

Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Efficient support for multicomputing on ATM networks. Technical report, University of Washington, Department of Computer Science and Engineering, 1993. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.2262.

Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. Practical performance models for complex, popular applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pages 1–12, New York, NY, USA, 2010a. ACM. ISBN 978-1-4503-0038-4. doi: 10.1145/1811039.1811041. URL http://doi.acm.org/10.1145/1811039.1811041.

Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. Practical performance models for complex, popular applications. *ACM SIGMETRICS Performance Evaluation Review*, 38(1): 1–12, June 2010b. ISSN 0163-5999. doi: 10.1145/1811099.1811041. URL http://doi.acm.org/10.1145/1811099.1811041.

Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. Empirical comparison of regression methods for variability-aware performance prediction. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 186–190, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3613-0. doi: 10.1145/2791060.2791069. URL http://doi.acm.org/10.1145/2791060.2791069.

Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 39–50, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4404-3. doi: 10.1145/3030207.3030216. URL http://doi.acm.org/10.1145/3030207.3030216.

Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. Transferring pareto frontiers across heterogeneous hardware environments. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, pages 12–23, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450369916. doi: 10.1145/3358960.3379127. URL https://doi.org/10.1145/3358960.3379127.

VideoLAN Organization. x264, the best H.264/AVC encoder. http://www.videolan.org/developers/x264.html, 2020a. Accessed April. 15th, 2016.

VideoLAN Organization. x264, the best H.264/AVC encoder. http://www.chaneru.com/Roku/HLS/X264_Settings.htm, 2020b. Accessed April. 15th, 2016.

Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 190–199, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2. doi: 10.1145/2351676.2351703. URL http://doi.acm.org/10.1145/2351676.2351703.

Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, Washington, DC, USA, May 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.32. URL https://doi.org/10.1109/FOSE.2007.32.

Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. Performance prediction of configurable software systems by fourier learning (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 365–373, Washington, DC, USA, November 2015. IEEE Computer Society. ISBN 978-1-5090-0025-8. doi: 10.1109/ASE.2015.15. URL https://doi.org/10.1109/ASE.2015.15.

Yi Zhang, Jianmei Guo, Eric Blais, Krzysztof Czarnecki, and Huiqun Yu. A mathematical model of performance-relevant feature interactions. In *Proceedings of the 20th International Systems and*

Software Product Line Conference, SPLC '16, pages 25–34, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4050-2. doi: 10.1145/2934466.2934469. URL http://doi.acm.org/10.1145/2934466.2934469.