Improving System Performance via Design and Configuration Space Exploration

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Chong Tang

December 2018

Abstract

The runtime performance of complex software systems often depends on the settings of a large number of static system configuration and design parameters. For example, big data systems like Hadoop present hundreds of configuration parameters to engineers. Many of them influence runtime performance, and some interact in complex ways, which make the configuration spaces for such systems large and complex. It is hard for engineers to understand how all these parameters affect performance, and thus it is hard to find combinations of parameter values that achieve available levels of performance. The result in practice is that engineers often just accept default settings or design decisions made by tools, leading such systems to underperform significantly relative to their potential. This problem, in turn, has impacts on cost, revenue, customer satisfaction, business reputation, and mission effectiveness. To improve the overall performance of such systems, we addressed three sub-problems.

The first is that we lack adequate concepts, methods, and tools to do tradeoff space analysis to find high-performing designs, where performance is assessed in multiple dimensions, particularly time and space utilization. To address this problem, we conducted the first systematic evaluation of an approach to finding Pareto-optimal system designs in such tradeoff spaces previously developed by Bagheri et al. This approach starts with relational logic specification of a system for which a design is to be found. This specification is conjoined with a separate specification of the mapping from system specifications to tradeoff spaces, comprising sets of designs, each consistent with a given specification. Second, this approach uses additional synthesis techniques to implement each design in

the tradeoff space as a running system. The framework that we developed then uses large-scale distributed computing based on Spark to parallelize the profiling of each such implementation using a synthesized benchmark test suite to measure its relevant performance characteristics. Our framework then selects the subset of Pareto-optimal designs for presentation to the engineer, who makes final tradeoff decisions. Building on the prior work of Bagheri et al., we evaluated the feasibility and benefits of the tradeoff space analysis approach with experiments in the domain of object-relational mapping (ORM) problems. The challenge, given an object-oriented (OO) data model, is to find a relational database schema for storing and searching such data efficiently. In general, there are many such schemas, with varying performance characteristics. Our results show that our approach can find schemas for OO data models (with tens of classes and relationships) that significantly outperform, in both time and space performance, those produced by widely used ORM tools, such as Ruby on Rails and Django. This work demonstrates the potential for formal tradeoff space synthesis and profiling to significantly improve the performance of component-level design problems in industrially meaningful systems.

Second, we lack approaches for developing accurate models for predicting the performance of highly configurable big-data systems a function of their configurations and the complexity of jobs running on them. Having such predictive models could significantly reduce the cost of finding good configurations for such systems. This is a hard problem because such mappings are not straightforward in general. Our overall approach is to derive such models by applying machine learning techniques to training data obtained from industrial big-data systems specifically Hadoop. The particular problem that we addressed is that learning such mappings tends to produce inaccurate models. Our approach was to find new ways to eliminate noisy values from training data. The first step is simply to ignore parameters that could have no bearing on performance. Our additional novel technique was to further exploit semantic meanings of configuration parameters and the job complexity to eliminate noisy values. Our results show significant improvement in prediction accuracy of learned models with such *semantically cleaned* data. We evaluated our approach by building

models for the Hadoop MapReduce framework, with and without using our semantic data cleaning approach on data obtained from Walmart Labs. In our experiments, our approach improved model accuracy from under 0.75 to over 0.88 for CPU time prediction. These results suggest that semantic cleaning of data provides real value for training performance prediction models for large big-data systems.

The third problem that we addressed was to find high-performing configurations for systems like Hadoop without the need to learn generalized performance prediction models. Learning such models is costly. Moreover, we observed that industrial uses of such systems tend to under-sample configuration spaces, making it hard to learn generalized models from such data. The challenge we addressed is to find a more effective method: one that doesn't rely on predictive models at all. Our overall approach is to employ a direct meta-heuristic search of performance-relevant configuration parameter values. For each such configuration, we profile a running system using the well known HiBench benchmark suite. Novel aspects of our approach include: 1) the use of evolutionary Monte Carlo Markov Chain configuration space search technique; 2) profiling performance using HiBench *small* datasets as proxies for the performance objective function for much larger datasets; and 3) validating the usefulness of configurations found for one MapReduce problem on classes of problems with similar resource usage patterns as judged using operating system call monitoring tools.

We show that, at an acceptable cost, this combination of ideas can find configurations for complex systems that significantly improve their runtime performance. To evaluate our approach, we applied it in the domain of big-data systems, particularly Hadoop and Spark. We measured performance improvements achieved by our approach and by competing approaches by comparing performance using configurations discovered by these approaches against performance using the widely employed *default* configurations distributed with these big data systems. We compared our approach against three competing approaches by comparing the extents to which they improved system performance: 1) a naive random search strategy; 2) a basic genetic algorithm; 3) and the cutting-edge, model-learning-based approach of Nair et al. Our approach improved the performance

of five canonical Hadoop jobs by 14% to 25%, relative to default configurations, and of another five Spark jobs by 2% to 17%, significantly outperforming the three competing approaches. These experimental results support the claim that our approach has significant potential to improve the performance of industrial big-data systems.

This dissertation solves three problems to improve system performance with novel software synthesis + distributed large-scale profiling, learning with parameters' semantic meanings, and meta-heuristic searching combined with cost-saving tactics. The results show that we can significantly improve system performance in critical engineering domains, such as ORM design and big-data infrastructures tuning.

Acknowledgments

The six years at the University of Virginia were the most precious time of my life, but it was not so easy. I could not finish this long journey without support, advice, devotion, and sacrifice of many people. I would like to express my gratitude to them in this section.

First of all, my sincere gratitude goes to my advisers, Kevin Sullivan and Baishakhi Ray. I am fortunate to have them as my advisers. They gradually trained me with insightful ideas and questions, rather than just providing answers. They were patient, always stood with me and encouraged me. Working with them was enjoyable and wonderful learning experience. They are not only my research advisers but also great mentors in my life. I still remember how Kevin helped when I had a car accident in the first year of my Ph.D. study, and how Baishakhi encouraged me with her own experience in many difficult times. This dissertation would not be possible without their help.

I would like to thank Alfred C. Weaver for serving as the chair of my dissertation committee, and Mary Lou Soffa, Joanne Bechta Dugan, Jonathan Bell for serving as my committee. They gave me great and constructive advice on my research and career development.

For the last six years at UVA CS, I am so grateful to have many friends who are humorous, sophisticated, smart, and hard-working. They are Yue Liu, Xi, Ke, Chunkun, Lin, Ritambhara, Dezhi, In Kee, Chi, Yong, Haoran, Essex, and so many friends that I cannot list them all. Thank you all for your friendship and support. I would like to thank my brothers and sisters in Christ, Diheng, David, Rachel, Timothy, Yunge, Qing, Yinping, and many others. Thank you for your pray and encouragement.

I would never be able to accomplish this without endless support from my family. Thank you, my parents Zibao Tang and Sulan Wu, and my uncle Shaoen and aunt Lingyan, for their love, support, and advice during this long journey.

At last, I thank my funding resources NSF CCF-1619123 and CNS-1618771 for their financial support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software performance is critical in many aspects of our lives. Slow applications can cause revenue loss, deterioration in customer satisfaction and brand reputation, and mission effectiveness. However, it is difficult to design and tune complex systems that substantially achieve their performance potential because runtime performance is determined by a large number of design and configuration parameters, the tuning of which is beyond the capabilities of most system designers and implementers. For example, big data systems like Hadoop [1] present hundreds of configuration parameters to engineers. Many of them affect performance, and some of them interact in complex ways. The available evidence suggests that most users of these systems make little effort to tune them beyond their default settings [2].

1.1 Problems

The large number of parameters and their interactions create three problems in software development and usage:

- **Designing high-performing systems**. It is difficult to find system designs that meet both functional and performance requirements given

system specifications that often under-specify non-functional properties. We say such a specification is incomplete.

- **Predicting system performance**. It is hard to predict the runtime performance of given configurations of a complex system due to parameters often interacting with each other in complex ways.

- **Configuring system for high performance**. It is hard to find a configuration that can yield high performance for complex configurable systems. The result again is that engineers often simply accept the default settings, which leads to significantly low performance compared to systems' potential.

This dissertation develops and evaluates a set of methods for dealing with such problems, mainly focused on exploring design and configuration spaces of software systems to enable them to operate efficiently.

1.2 State-of-the-art

This section summaries previous work on how researchers and engineers currently solve the problems mentioned above.

Formal Synthesis and Analysis of System Designs. The state-of-the-art work starts to adopt formal synthesis in finding system designs given incomplete specifications and objective functions. Dang [3] provided a tool for embedded software synthesis. Le [4] provided a framework to synthesize programs for data extraction from different kinds of sources. Assayed [5] provided a synthesizer for multi-threaded software on multi-processor architectures. Srivastava et al. [6] developed a proof-theoretic synthesizer, which accepts user-provided relations between inputs and outputs of a program and generates a constraint system such that solutions to that set of constraints yields the desired program. Sketching [7] is also a synthesis technique in which programmers partially define the control structure of the program with holes, leaving the details unspecified.

This set of work has two main shortcomings in the context of improving runtime performance. First, most of it focuses on functional requirements not on non-functional properties, such as runtime performance. Second, it relies on user input, like user-provided relations between inputs and outputs or control flow, to synthesize a system. Our work shares the common insight with this work on incomplete specifications and synthesis based on constraint solving. Given an incomplete specification which often defines only functional properties but under-specifies non-functional properties, we provide fully automated synthesizers that can sample many or in some cases all designs and general a common set of performance test cases. We then conduct a large-scale dynamically profiling to figure out the runtime performance of all designs and then find Pareto-optimal instances. The final designs are those satisfying the given specification with often greatly improved runtime performance.

Learning Better Performance Models. To solve the second problem of predicting the performance of a given configuration, researchers aim to learn generalized predictive models from historical data [8–11]. Most previous work mainly treated such data as pure numbers and focused on things like how to select features, and how to transfer the learning problems to another format so that one can build highly accurate models. Zhang et al. [12] consider the case where all configuration parameters are independent boolean variables. They formalized the performance prediction problem as learning the Fourier coefficients of a function f. A shortcoming is that their approach can only predict system performance for configurations containing only boolean values. Song et al. [13] presented a way to predict performance for Hadoop jobs. They profiled job features by dynamically executing jobs on small sampled data sets. The disadvantage is that this approach works only on small sets of features: five and six specified features in the map and reduce phases, respectively, such as data conversion rates for Map and Reduce function. Due to the specific features related to time performance, their approach cannot be applied to predict other performance metrics like memory utilization. In this dissertation, we hypothesize that the semantic meanings of configuration parameters could be helpful to improve the accuracy of performance prediction models.

Searching Configuration Spaces for Performance. To find high-performing configurations for complex configurable systems, researchers also mainly rely on learning-based approaches [8–10, 14]. The main challenge of this trend of work is that the training data mainly determines how accurate a trained model will be. Many such approaches suffer from a lack of quality training data. For systems like Hadoop, whose users tend to stick with default settings [2], the obtained traces are likely not to be diverse enough to learn generalized models. Moreover, due to complex parameter interactions, it's difficult to sample a good set of training data. Nair et al. [11] argued that inaccurate models can also help find better configurations as long as they preserve the relative ordering of configurations by performance. This dissertation addresses this problem without trying to learn generalized performance models at all. Using meta-heuristic search algorithms, we sample configurations and profile their performance and use that information to guide the search process. We use two tactics to save the cost of searching and profiling. First, we profile a configuration with smaller jobs and verify its validity with 100X larger instances. Second, we hypothesize that a configuration found with one job could also yield similar performance for jobs that have similar resource utilization patterns.

1.3 Goal

Our goal is to show that we can expand the scale of tradeoff analysis and configuration space exploration and to make them operate efficiently with following approaches.

- Our first goal is to show that we can find Pareto-optimal system designs *w.r.t.* to time and space performance for industrial meaningful systems.

- Our second goal is to show that we can improve the accuracy of performance prediction models by leveraging the semantic meanings of configuration parameters of a complex system to clean the training data.

- Our third goal is to show that we can find significant better configurations for big-data infrastructures which sizes haven't been studied before.

1.4 Insights and Approaches

Our first insight is that, by leveraging current advances in formal synthesis and distributed computing techniques, we can find Pareto-optimal designs for given systems. We developed and evaluated a framework to find desired ORM schemas. We first formally specify a given ORM model in relational logic embedded in Alloy [15]. Then, we use a relational constraint solver to synthesize all designs and a set of common test loads. Next, we dynamically profile all designs. Finally, we select Pareto-optimal designs based on profiling results.

Our second insight is that semantic meanings like the dependent structure of configuration parameters, where the relevance of one parameter can depend on the value of another, can help eliminate spurious information from training data and can improve the performance of prediction models. To evaluate this idea, we first collected a broad set of usage data from a commercial Hadoop cluster. Then, we cleaned the data with parameters' dependency structure. Next, we build two prediction models with datasets cleaned with and without such semantic meanings. Finally, we compared the quality of two models to show the improvement.

Our third insight is that we can directly search a configuration space with heuristic algorithms, without learning expensive and inaccurate predictive models at all. In this dissertation, we use an evolutionary Monte Carlo Markov Chain (EMCMC) algorithm to search Hadoop and Spark's configuration spaces for high-performing configurations. We also use two strategies to save the cost of searching.

1.5 Thesis Statement

I claim that our approaches can deal with large and complex systems and can enable design and configuration space exploration to operate practically and efficiently.

- First, I claim that our tradeoff space analysis approach can find Pareto-optimal designs *w.r.t.* performance objective functions and can outperform widely-used ORM tools and methods that people use in practice.
- Second, I claim that the semantic meanings of configuration parameters can improve the accuracy of performance prediction models.
- Third, I claim that our configuration space exploration approach based on heuristic search can find high-performing configurations for large and complex software systems and can outperform baseline search algorithms and cutting-edge learning-based approaches.

1.6 Evaluation Experiments

To evaluate our first approach, we compare it with two widely used ORM tools, namely Ruby on Rails and Django. We studied seven OO data models from the literature and real systems. For each one, we specify it in relational logic using Alloy. Then we synthesize all schemas (obviously only for modest-scale models) using the Alloy model solver. Next, we profile all schemas with our Spark-based framework using synthesized and randomly generated test suites. We measured the performance of all schemas on two common database management systems: MySQL and PostgreSQL. Finally, we select Pareto-optimal designs and compare their performance with those generated by Ruby on Rails and Django. We studied how much performance improvement Pareto-optimal schemas can gain comparing to the solutions of Rails and Django and found it to be significant.

To evaluate our second approach, we collected a broad set of logs of Hadoop MapReduce jobs started with Hive queries from an industry cluster of Walmart

Labs. We identified the dependency structure of Hadoop parameters based on the official documentation. Then we cleaned the data using such semantics, and we trained two random forest models using datasets with and without our cleaning approach applied. Finally, we compared the R^2 and cross-validation scores as the indicators of model accuracy.

To evaluate the third approach, we built our search algorithm based on EMCMC, and compared it with three baseline approaches: 1) a basic genetic algorithm, 2) a purely random search algorithm, and 3) a cutting-edge model-learning-based approach of Nair et al. [11]. We selected Hadoop and Spark as the study systems. For each system, we first studied the performance improvement of configurations found by our approach over the default configurations shipped with Hadoop and Spark. Then, we ran all three baseline approaches to see how much performance improvement they can obtain over the default configurations. At last, we studied how good our approach performed compared to the baselines.

1.7 Results and Interpretation

All three approaches achieved good results in the domains and systems that we studied. In particular:

- The first set of evaluation results in the ORM domain shows that our tradeoff space synthesis + large-scale profiling and selection found Pareto-optimal designs that have 30% to 70% better time and space performance than the ORM tools packaged with Ruby on Rails and Django, on both MySQL and PostgreSQL DBMSs.

  This results support our claim that our formal synthesis and profiling approach has the potential to outperform today's widely used tools.

- The second set of evaluation results in MapReduce performance prediction shows that semantic meanings of configuration parameters can help improve the model accuracy of performance models from under 0.75 to over 0.88.

This suggests that parameters' semantic meanings do have real value in improving the performance of prediction models.

- The third set of results in searching high-performing configurations for Hadoop and Spark shows that we can gain 14% to 25% more performance for studied jobs over the default configuration. Moreover, our approach yielded performance improvements of 7% to 125% over a random search algorithm, 6% to 85% over a basic genetic search technique, and 5.4% to 1,700% over a cutting-edge model-learning-based approach.

  These results strongly suggest that meta-heuristic algorithms can improve the performance of large and complex big data systems.

Chapter 2 reviews the background of this dissertation and discusses related work at a high level. Chapter 3, 4, and 5 present these three approaches and corresponding experiments and evaluation results in detail. Chapter 6 evaluates the whole work, and Chapter 7 concludes this dissertation.

Chapter 2

Background

We review related work at a high level in this chapter. We will discuss particularly relevant related work in greater detail in later chapters.

2.1 Software Synthesis

Our first design space analysis approach spans and leverages techniques from software synthesis and tradeoff analysis. Srivastava et al. [6] developed a proof-theoretic synthesis method, in which the user provides relations between inputs and outputs of a program in the form of logical specifications, specifications of the program control structure as a looping template, a set of program expressions, and allowed stack space for the program to be synthesized. Their work then generates a constraint system such that solutions to that set of constraints yields the desired programs. They have shown the feasibility of their approach by synthesizing implementations for several algorithms from logical specifications. Sketching [7] similarly is a synthesis technique in which programmers partially define the control structure of a program with holes, leaving the details unspecified. This technique uses an unoptimized program as correctness specification. Given these partial programs along with correctness specification as inputs, a synthesizer, based on SAT solver, is then used to complete the low-level details to complete

the sketch by ensuring that no assertions are violated for any inputs. This work shares with ours the common emphasis on incomplete specifications, synthesis based on constraint solving, and optimization under an objective function.

Another work [16] provided a high-level synthesis framework to transfer a behavioral description, a functional specification, in ANSI-C to register-transfer level VHDL with parallel compiler transformation technique. They also integrated heuristics based on mutual exclusivity of operations, resource sharing, and hardware cost models.

The commonality of our work and that discussed above is that they all need user-provided information, such as system inputs and outputs or control- and data- flow, to guide the synthesis logic. The synthesis here refers to the concrete example of correct or incorrect behavior to prune the design space, and finally narrow down the design space to one implementation that satisfies the given specification. This early prune is a good way to narrow down the search space. However, it could also eliminate many high-performing solutions.

Different from all these techniques, our approach tackles the automated tradeoff space analysis through synthesizing spaces of design alternatives and optimizing using common measurement functions over such spaces. It thus relieves the tedium and errors associated with earlier more manual approaches.

2.2 Performance Modeling

In the system performance prediction area, one of the biggest problems is that it is hard to build highly accurate performance models. The main reason is that modern software systems typically have extensive configurability. For example, the Linux kernel has 6918 configuration options in version 2.6.33.3 according to Liebig *et al.* [17]. Hadoop has more than 900 parameters in version 2.7.3. Such dimensions give rise to vast configuration space. Besides, it is very possible that configuration settings interact with each other with respect to performance, making search for high-performing states a complex task. Most recent work focuses

on how to reduce the dimension of a configuration space through automatic feature selection. In work on performance influence models [14], Siegmund *et al.* used step-wise linear regression to derive a performance-influence model for a given configurable system which can provide users the description of possible impacts among all configuration options and their interactions. They added the features hierarchically to the learning algorithm. Their approach only considers binary and numeric configure options. One shortcoming is that we do not know how parameters interact with each other. It could be two-way, or three-way, or of even higher of arity. Step-wise linear regression cannot catch such interactions.

In our project, we used the domain knowledge to select features that influence the performance of a given execution. We selected Hadoop as the study system. We first studied the dependent structure of all parameters and then reduced this information to clean the training data. In particular, we would ignore data that the settings of other parameters made irrelevant. We also referred to previous work as reviewed and summarized in [18], in which all options are thoroughly studied. This set of work gives us enough information to determine the features that are performance-related and their dependent structure. Based on such information, we grouped all configuration options by sub-systems and incorporated them one-by-one in a hierarchical model.

2.3 Searching for Better Configurations

Many engineering and applied science systems have complex configuration spaces. It is not easy to find high-performing configurations for such systems. Therefore, configuration space exploration with meta-heuristic search is a common approach in many engineering domains. Zhang *et al.* [19] use on-chip design space exploration to solve the problem of unnecessary flushing and energy waste. They search a 4-parameter space in hardware, whereas we work on much larger spaces. Weimer *et al.* [20] uses genetic programming to find test-passing code snippets to reduce the cost of manual software debugging. It is a compelling

proof of concept work that broke new ground. However, the correctness of this approach is unsound and relies entirely on the test suite as a de facto objective functions. Baltz *et al.* [21] uses a human-guided meta-heuristic search to find high-performing configurations for plasma fusion experiments. The configuration spaces we work on have similar dimensionality, but we also reduce the exploration cost by 100X yet often find good configurations.

2.4 Missing Gaps and Fixes

Most prior art discussed above does not consider the complexity of design and configuration spaces, treating them as simple product spaces: "A design space is a product of possible design choices" [22]. However, in practice, a design space is a subset of the cross-product of design/configuration parameters: a relation of design parameters satisfying *constraints*, which makes the structure of a design/-configuration space much more complex than often assumed. Moreover, most previous approaches only work on small and simplified spaces, as exemplified by work with some thousands of configurations as seen in [11]. By contrast, modern software systems often have hundreds or even thousands of parameters, making the sizes of such spaces vast. Applying above methods on such systems would create high costs. Our first approach synthesizes all satisfying designs, albeit with similar scale limits, and then conducts large-scale profiling to find Pareto-optimal designs in the space, while our second approach leverages constraints among parameters (dependent structure) to improve accuracy in system performance prediction modeling. In the third approach, we use two tactics, namely which we refer to as scale-up and scale-out, to vastly reduce the cost of meta-heuristic configuration space exploration.

The next three chapters will discuss these issues in detail and present solutions and evaluations.

Chapter 3

Design Space Exploration to Improve System Performance

This chapter presents our approach to using formal synthesis and large-scale profiling based on parallel and distributed computing to find high-performing system designs based on incomplete specifications, constraints, and objective functions. We present a set of evaluation results from applying our approach in the ORM domain to find high-performing database schemas.

Developing software systems that achieve different kinds of performance requirements, balancing various tradeoffs, remains a major engineering problem. Developers usually start with a specification, like a detailed design document or a "story" in a project management system, and implement a version based on their understanding of the specification. However, specifications are generally incomplete concerning the totality of desirable system properties, especially non-functional properties, many of which often depend on the choice of design. For example, while an object-oriented data model, as a specification, constrains the behavior of a persistent data store, it does not uniquely determine the database

schema (design); nor does it express preferences for performance properties or tradeoffs involving read and write times and file sizes, which, in general, can vary significantly with the choice of schema.

Unspecified or under-specified properties create degrees of freedom that give rise to *tradespaces*: sets of designs that satisfy a given specification but that vary in other relevant properties, again often non-functional properties such as time and space performance, reliability, and evolvability. An engineer then has two choices. The first is what we might call a *single-point strategy*: use design heuristics, tacit knowledge, or even just training and experience to select some design that satisfies the explicit specification without systematically considering tradeoffs in dimensions on which the specification is silent. Such methods develop point solutions in the hopes they will be good enough for stakeholders in all key dimensions. In fact, when tools automatically produce implementations, they often use *single-point* strategies. Consider object-relation mapping (ORM) tools, now provided in many programming environments. They map object-oriented data models to relational schemas and code for managing application data. They often use a single mapping strategy, just producing some result, and do not systematically aid engineers in understanding the space of possible solutions or the tradeoffs in time and space performance that they entail.

Recently, engineers are focusing less on point solutions and more on specifying, populating, and analyzing points and regions in design spaces. The contribution of this work is an approach to the specification-driven synthesis of both design spaces and of common performance benchmark test loads for fair, comparative, dynamic analysis of non-functional properties of variant designs across such spaces. It is a challenge to synthesize test loads for fair comparative analysis because doing so generally requires specialization of common loads to the variant interfaces of variant designs. Our goal is to synthesize both design spaces and such loads automatically from common, formal, design-space specifications, to enable specification-driven automated dynamic analysis of tradeoffs among non-functional properties across large design spaces.

The alternative to single-point design is a *tradespace analysis strategy*, in which

one considers design spaces explicitly, estimates or measures critical properties of many alternatives in such spaces, then choose one that realizes a desirable set of properties, often by optimizing with respect to an objective function that ultimately includes developer preferences for how to make final tradeoffs on a Pareto frontier. The fundamental *hypothesis* driving the development and use of systematic and automated tradespace analysis is that *it often produces results of greater value than single-point design even net of its additional costs.* Indeed, tradespace analysis (TA) tends to reveal designs that otherwise engineers might miss.

This chapter presents our tradespace analysis strategy, particularly for design space models expressible in a relational logic [15], to find optimal system design under given performance objective functions and tradeoff preferences. This approach leverages recent advancements in several areas, such as automated reasoning in relational logic, formal synthesis, and scalable big data analytics, to develop such a novel technology base. More specifically, this chapter contributes a formal, general theory of design space tradeoff analysis tools, and a map-reduce-based framework, derived from the theory, for implementing such tools. The theory is organized as a hierarchy of Coq typeclasses. From this theory, we derive a polymorphic framework (in Scala) that developers specialize and extend to produce domain-specific tradeoff analysis tools.

Our evaluation strongly suggests that our tradespace analysis approach enables the production of database designs that significantly outperform those produced by widely-used industrial ORM synthesis tools, such as Rails and Django, yet are entirely overlooked by such widely-used, industrial tools. To summarize, this work makes the following contributions:

- *Tradespace Analysis:* We present a tradespace analysis approach that can generate and evaluate all possible concrete designs given an (modest-scale) specification.

- *Theoretical framework:* We develop a theoretical framework conceptualized in constructive logic to establish a formal architecture for a generic tradeoff analysis tool.

- *Tool implementation:* We show how to exploit the power of our formal abstractions by building a mechanically derived, polymorphic implementation framework for tradeoff analysis tools.

- *Empirical evaluation:* We present our experiences with an evaluation of our approach in the context of tradeoff analysis of object-relational database mappings, the results of which support the claim that our approach reveals designs that significantly outperform those produced by widely-used, industrial ORM tools.

Section 3.1 presents object-relational mapping (ORM) as a concrete driving problem. Section 3.2 presents a mathematical and solution framework. Section 3.3 details an implementation architecture using a relational constraint language (Alloy) and solver (Alloy Analyzer) for expression and synthesis of design spaces and loads. Section 3.4 presents our experiments in the ORM domain and answers questions about the validity of our approach. Section 3.5 presents the evaluation results of our approach. To solve the problem of large design spaces, section 3.6 describes the detailed design and implementation of our automated design evaluation framework. Section 3.7 presents the results of a set of empirical studies using our distributed design space exploration framework. The rest of this chapter presents our evaluation of this work, related work, and conclusions.

3.1 Motivation

In this section, we discuss how incompleteness in specification gives rise to important separations of concerns and the need for a systematic understanding and application of tradeoff analysis to find optimal designs. We then present a running example having to do with tradeoff analysis among the space of possible design alternatives.

3.1.1 Incompleteness in Specification

Specifications are often incomplete with respect to the full range of properties that stakeholders value in a given system. Such incompleteness is often not a flaw. Rather, it can serve a strategic function in structuring the process of complex system design. When a specification is silent on system properties relevant to stakeholders, it partitions the design process, the representation of acceptable solutions, and the set of design decisions to be made. We address each of these concerns and explain how they create a need for a better theory of and technology for tradeoff analysis.

Partitioning of the Design Process. Incompleteness partitions the design process into at least two distinct parts. The first is a deductive process, in which candidate solutions are derived from a specification, constrained only by the condition that they satisfy its terms. Such solutions generally differ in stakeholder-relevant properties on which the specification was silent. The second part is thus an optimization process, in which solutions are evaluated for additional properties, tradeoffs are identified, candidates are ranked, and one is selected for use or development.

Partitioning of Design Representations. This deductive vs. optimization partitioning of the design process is mirrored by an explicit vs. implicit partitioning of the representation of what constitutes an acceptable solution. The explicit part is given by the specification. The implicit part is represented in the property estimation functions that will be used to evaluate solutions, the stakeholder utility functions (emergent or documented) that map property estimates to stakeholder utilities, and the stakeholder tradeoff functions (emergent or documented) that map the multiple stakeholder utilities to a final ranking of, and ultimately to a choice from among, candidate design solutions.

Partitioning of Design Decision Spaces. The deductive vs. optimization and explicit vs. implicit dichotomies extend to a split between decisions that are understood and agreed on well enough to be pinned down in a specification, and those that are not. This is a split between settled vs. unsettled decisions. A

specification speaks explicitly on design decisions that are settled while remaining silent on relevant but unsettled aspects, leaving them to be worked out in downstream, optimization-oriented design activities.

These separations of concerns can also sometimes be seen in the evolutionary dynamics of complex systems. As initially unsettled concerns are settled, they can migrate from being represented implicitly in measurement and utility functions to being explicit in specifications. System architectures can be seen as settled and explicit specifications, for example, that remain incomplete in other key areas. As optimization-based processes produce knowledge and agreement, these results can migrate into specifications.

3.1.2 Running Example

Our ultimate goal is to find desired designs with respect to performance for all kinds of systems that can be expressed in relational logic through advances in design space science and technology. To motivate this research and illustrate our approach, here we provide a running example having to do with tradeoff analysis among the space of possible solution alternatives. Here we use the analysis of spaces of object-relation mappings (ORM) as a tractable and useful *driving problem*.

In the ORM domain, a specification is an object model that contains objects and relations among these objects. Object-oriented data models serve as specifications for application database schemas. While these specifications constrain schemas, they are silent on properties such as time and space performance, and evolvability. At the same time, degrees of freedom in ORM mappings (e.g., in how inheritance is mapped to relations) give rise to spaces of satisfying schemas that vary in these properties. Object-oriented specifications of data models are incomplete regarding these other properties. While all solution points within the solution space are equally good at satisfying the specification, they will differ in other properties of interest in design. A developer starts with an object model as in Figure 3.1 and eventually selects one of many possible strategies for mapping

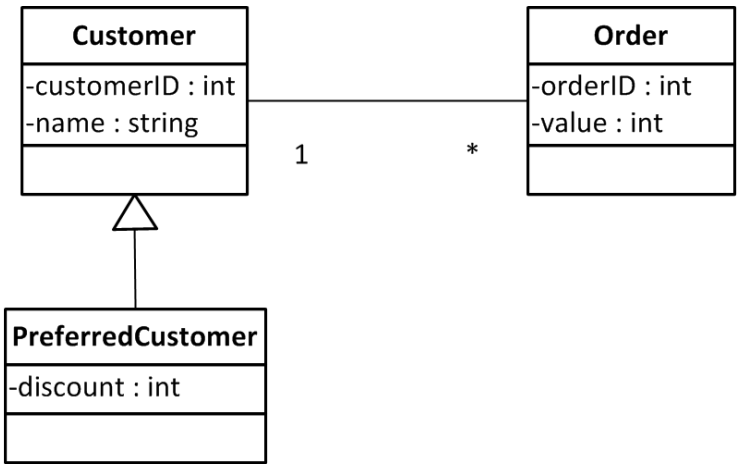

Figure 3.1: A simple object model with three classes, *Order*, *Customer*, and *PreferredCustomer*, a one-to-many association between *Customer* and *Order*, and with *PreferredCustomer* inheriting from *Customer*

such a model to a relational database, with tradeoffs in response time, space usage, evolvability. Figure 3.2 illustrates two such strategies; and Listing 3.1, a database set-up script derived from one of these solutions.

Simple ORM solutions, many in everyday use, lock one into either point solutions or highly constrained solution spaces. To address this problem, the work of Bagheri et al. [23] presented a capability to synthesize comprehensive ORM design spaces from formal object model specifications. Given such a space, the challenge is to develop, validate, and apply non-functional property prediction (*analysis*) functions to designs in the space to predict properties of designs in, and tradeoffs across, the space.

Ideally, one has a vector of easily computed, well-validated performance analysis functions. In that case, mapping this vector of functions (or, equivalently, this vector-valued function) across the points in the space yields a multi-dimensional, non-functional property *image* of the design space. Tradeoffs, Pareto-optimal solutions, and other critical information can then be read from the results.

Static analysis functions predict properties from the structures of design representations without actually implemting and testing the designs. Such functions are often efficient, but might not be available, validated, or sufficiently predictive. This point is clear in our earlier work. Cognizant of the issues, we applied

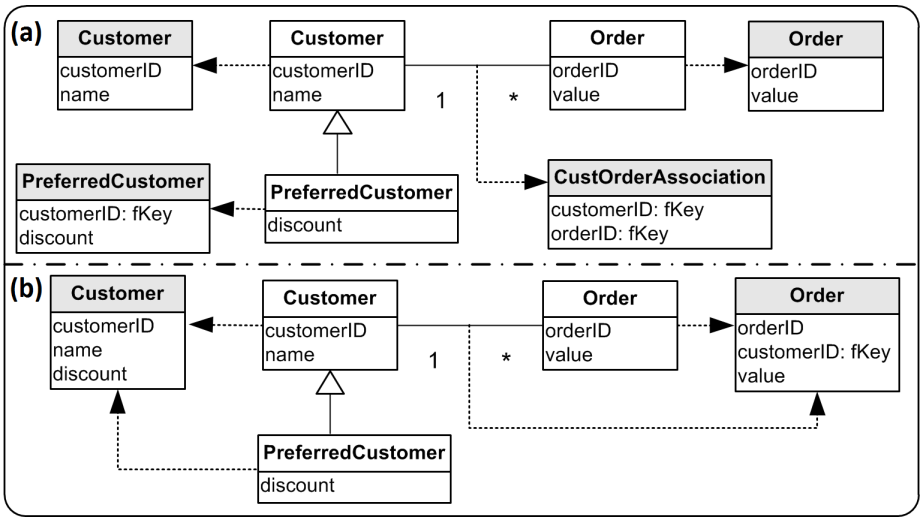

Figure 3.2: Two mapping strategies. White boxes represent classes; gray titles, corresponding tables, and black and white arrows, mapping and inheritance relationships. Foreign keys are marked as *fKeys*.

```
1  CREATE TABLE `Order` (
2  `orderID` int(11) NOT NULL,
3  `orderValue` int(11),
4  `customerID` int(11),
5  KEY `FK_customerID_idx` (`customerID`),
6  PRIMARY KEY (`orderID`)
7  );
8
9  CREATE TABLE `Customer` (
10 `DType` varchar(31),
11 `discount` int(11),
12 `customerID` int(11) NOT NULL,
13 `customerName` varchar(31),
14 PRIMARY KEY (`customerID`)
15 );
```

Listing 3.1: Synthesized MySQL database creation script elided for space and readability.

published but not very well validated static metrics [24, 25] to our synthesized ORM spaces. The results shows that while these predictors had some value, they were not very reliable. Can design decision makers believe such metrics? What are the actual properties and tradeoffs?

These questions took on added urgency with our production, reported here for the first time, of a web-accessible tool that implements our ORM synthesis and analysis approach. We call it Trademaker-ORM[1] (T-ORM). It supports automated, specification-driven synthesis of ORM design spaces and static analysis using the aforementioned metrics. It provides a web interface, user account and job management (job submission, asynchronous execution, status

[1]Available at www.jazz.cs.virginia.edu:8080/Trademaker at the time of this writing

reporting, persistence), computation and presentation of Pareto-optimal subsets of synthesized designs under the given metrics, and synthesis of SQL databases for selected designs. Figure 3.3 presents a screen-shot of a T-ORM run. Rows present Pareto-optimal designs, and columns, analysis results. Listing 3.1 presents an SQL database creation script obtained by selecting a design from the table.

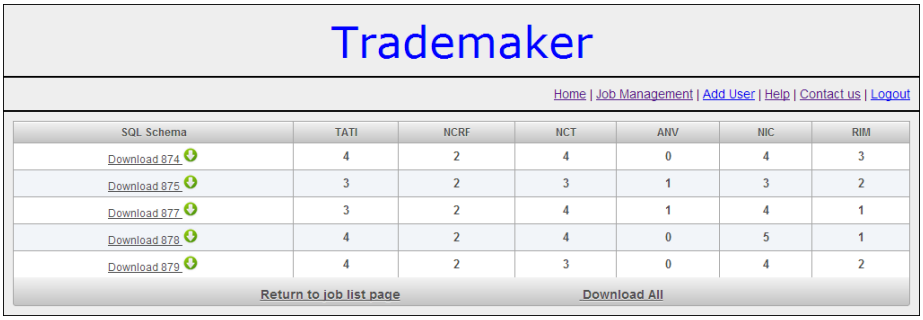

Figure 3.3: A view of our tool to provide decision-makers with Pareto-optimal OR mapping solutions based on static analysis results; columns and rows represent metrics and solution alternatives, respectively.

To test the validity of the results that T-ORM reports, we turned to dynamic performance analysis. The combination of ample computing capacity and our ability to synthesize many running databases for given object models suggested that we measure properties and tradeoffs of actual running systems, in the spirit of what Cadar et al. call *multiplicity computing* [26]. Doing so could at least validate the static metrics through statistical analysis of the power of these metrics to predict dynamically measured results. If we could validate the static metrics, we could use them for efficient analysis. If not, we could fall back on less efficient but more trustworthy dynamic analysis. In this case, a further question arises: how to optimize dynamic analyses. We were thus led to formulate three driving hypotheses for the research reported in the rest of this work:

- H1: The ordering of alternatives predicted by the static metrics predicts that of the dynamic analysis results
- H2: The relative magnitudes of static measures of alternatives predict those of the dynamic analysis results
- H3: Dynamic analysis using scale-limited synthesized loads predicts performance under much larger loads

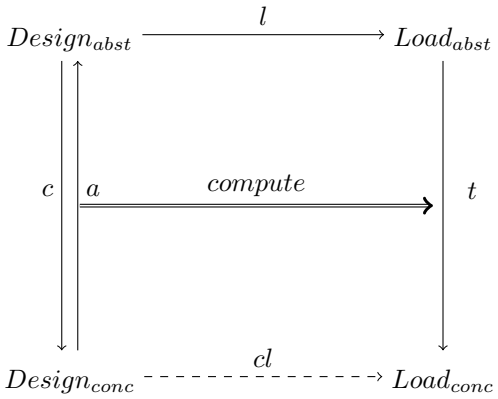

Figure 3.4: Algebraic structure of the Trademaker approach.

The problem was now to figure out how to automate *fair* comparative dynamic analysis of diverse database designs. There are many commercial tools for generating database testing loads from schemas. In our case, however, many variant schemas all implement a common object model. An application that operates against an object model will thus be evaluate against what we will call an *abstract* load. The translation from abstract load to a concrete operations on a particular database implementation emerges as a real challenge. We faced the need to synthesize thousands of load specialization functions.

3.2 Algebraic Model

The insight that enabled synthesis of functions mapping from abstract loads to concrete test loads specialized to individual schemas was that we could recover, from synthesized database designs, abstraction functions relating concrete designs back to the abstract object model specifications from which they were derived, and that from these abstraction functions we could derive functions for concretizing abstract loads synthesized from the same object models. The commutative diagram in Figure 3.4 presents the resulting mathematical structure. $Design_{abst}$ is a set of formal design space specifications in a particular domain, inductively defined by the grammar and semantics of the language in which the models are specified. In work to date, we represent design space specifications as expressions in a domain specific language embedded in a relational logic.

To an abstract model, $m \in Design_{abst}$, we apply a design space synthesis (concretization) function, c, to compute $c(m) \subset Design_{conc}$, the space of concrete design variants from which we want to choose a design to achieve desirable trade-offs. Relation c can be seen as mapping abstract, intensional models of design spaces to extensional representations, namely sets of concrete design variants. We represent the design space synthesis function, c, as a semantic mapping predicate in our relational logic, taking expressions in the abstract modeling language to corresponding concrete design spaces. Relation a, an abstraction relation, explains how any given concrete design, $d \in c(m)$, instantiates (i.e., is a logical model of) its abstract model, m. Function c is specified once for any given abstract modeling language, as a semantic mapping predicate in our relational logic.

Relation l, an abstract load generation relation, similarly maps an abstract model, $m \in Design_{abst}$, to a set of abstract loads $l(m) \subset Load_{abst}$.

To dynamically analyze a concrete design, $d \in c(m)$, an abstract load, $l_d \in l(m)$ has to be specialized for the particular design. The function t, a load concretization function, serves this purpose. We compute t from a. As long as $c(m)$ preserves a representation of a in its output, then from any single design space model, m, we can synthesize a concrete design space, and both abstract and concretized loads.

The derivation of t from a induces a mapping, cl, from concrete designs to concrete loads parameterized by a choice of abstract load. This function completes the commutative diagram. We do not actually implement this mapping. The next section describes instantiation of our approach for the particular domain of object-oriented relational database persistence mappings.

3.3 Model Implementation

This section presents an implementation architecture for our approach (for ORM). Figure 3.5 gives a high-level overview. Boxes represent processing modules, and

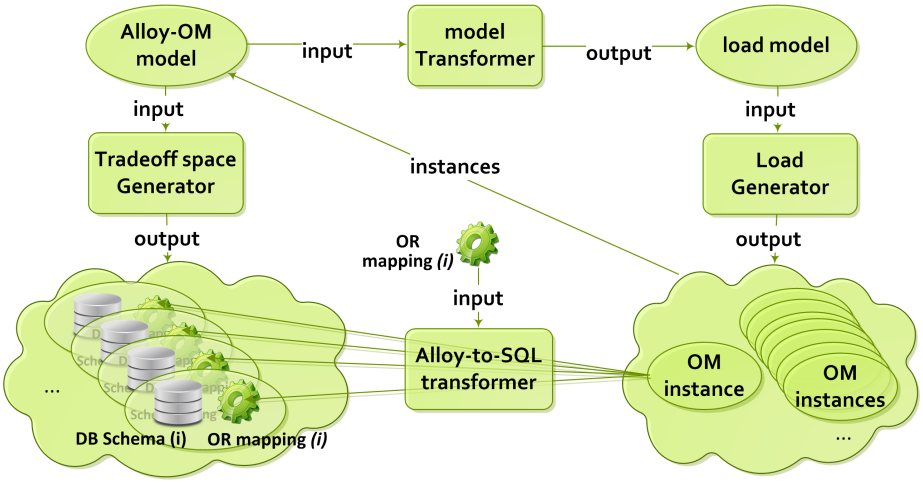

Figure 3.5: High-level overview of the Trademaker implementation for the particular domain of ORM.

ovals, module inputs and outputs. Abstract models are given as expressions in Alloy-OM, a domain-specific language embedded in Alloy's relational logic language. Roughly speaking, we specify c and l as predicates in Alloy. Conjoining these predicates to an Alloy-OM model yields a specification of the desired output space. The Alloy analyzer computes the results, encoded as Alloy models, which we then unparse into useful forms. From concrete design models we extract SQL database creation scripts. From abstract load models combined with choices of concrete databases, we derive concrete loads represented as sequences of SQL insert and select queries.

The rest of this section details how we implement the main components of the approach, corresponding to functions c, l, and t. Subsection 4.1 explains how we implement c, for synthesizing concrete models from abstract design space specifications; 4.2, how we implement l, for synthesizing abstract loads from the same specifications; and 4.3, how we implement t, for concretization abstract loads.

3.3.1 Design Space Synthesis

Mapping abstract object models to concrete designs was the subject of earlier work [23]. Our novel results since that work include substantial refinement and

validation of mapping rules and the production of functions that unparse concrete designs representations, encoded as Alloy objects, into SQL database creation scripts. This translation is important for tool users as a key to automating dynamic analysis.

This subsection briefly reviews the approach. We briefly describe relevant new work. Given an object model expressed in our Alloy-OM domain-specific language (which is just a set of definitions in Alloy), we use Alloy's constraint solver to synthesize concrete OR mapping strategies, realizing the function, c. The synthesizer uses our Alloy-encoded formalizations of best practices for object-relational mapping, described informally in the literature [27–29].

Alloy-OM supports three main constructs: *Class, Attribute* and *Association*. Each class in an object (e.g., UML) model appears as a *Class* signature in Alloy-OM. Each attribute of a given class appears as an Alloy-OM *Attribute* in the *attribute set* of the corresponding Alloy-OM Class signature. Each association in the object model appears as an Alloy-OM *Association* signature.

```
1 one sig Order extends Class{}{
2   attrSet = orderID + orderValue
3   id=orderID
4   isAbstract = No
5   no parent
6 }
```

Listing 3.2: Order class in Alloy-OM.

Listing 3.2 presents a fragment of an Alloy-OM model for our customer-order example. The order class has two attributes, "orderID" and "orderValue", assigned to the "attrSet" field of the Order class. The id field specifies the orderID as the identifier of this class. The last two lines of the Order signature specification denote that Order is not an abstract class and has no parent. Alloy-OM is not a sophisticated modeling language. The salient point is that embedding it in Alloy allows us to use Alloy relations to encode, and the Alloy analyzer to compute, formally specified semantic mapping rules to other domains: here concrete database designs and abstract loads for those designs.

Figure 3.6 presents a graphical depiction of an Alloy object encoding a synthesized OR mapping solution. This solution is one of five Pareto-optimal solutions in the design space for our customer-order object model. The diagram is accurate but edited to omit some details for readability. In this diagram, Table1 is associated

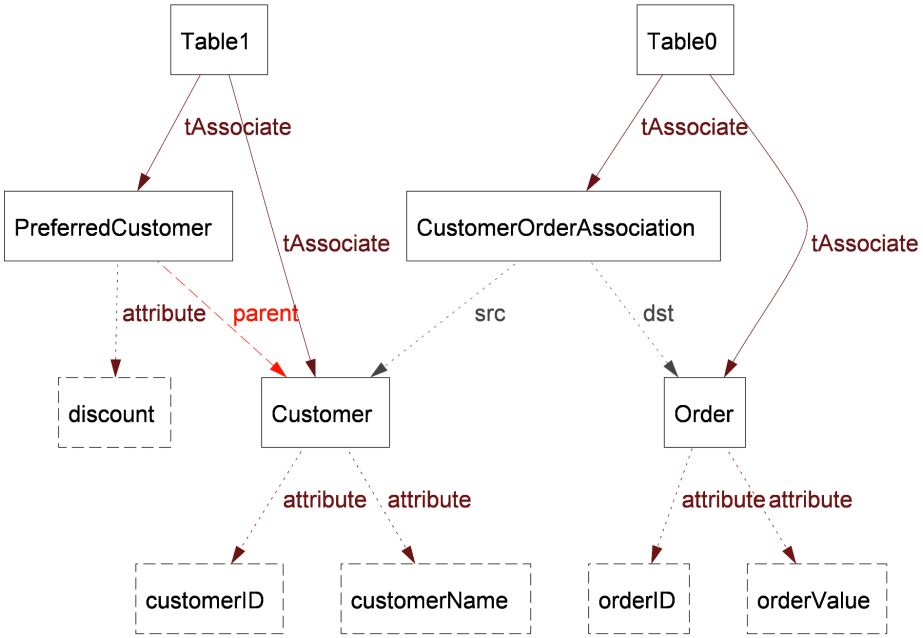

Figure 3.6: OR mapping for customer-order example

to Customer and PreferredCustomer classes, and Table0 is associated to both Order and CustomerOrderAssociation.

From this Alloy solution, our tools generate the SQL script of Listing 3.1. The script sets up a database with the two tables: *Order*, with attributes *orderID*, *customerID* and *orderValue*; and *Customer*, with attributes, *customerID*, *customerName*, *discount*, and *DType*. Both *Customer* and *PreferredCustomer* objects are stored in this table under this particular mapping strategy, with the *DType* field distinguishing the type of record stored.

3.3.2 Abstract Load Synthesis

Our approach to synthesizing abstract loads starts with the automated transformation of a given Alloy-OM model into a related Alloy specification that we call a load model. We then use the Alloy Analyzer to synthesize abstract loads from this load model. Alloy solutions to the load model encode abstract object model data instances (*OM-instances*), which are what we take as synthesized loads with which to test synthesized designs. This section describes this functionality in more detail.

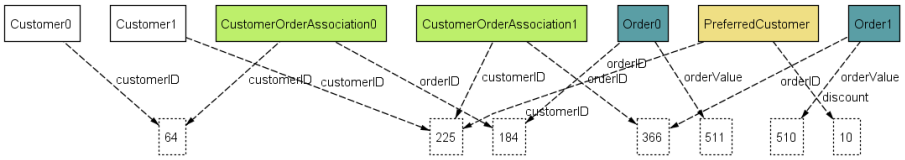

Figure 3.7: An example of OM-instance.

For each instance of *Class* and *Association* in the Alloy-OM model, our *model transformer* synthesizes a signature definition. When the class under consideration inherits from another class, the synthesized signature definition extends its parent signature definition. Given the specification of *Order* represented in Listing 3.2, the following code snippet represents its counterpart in a synthesized load model

```
sig Order{
  orderValue: one Int,
  orderID: one Int
}
```

The *one* multiplicity constraints used in the declaration of elements' signatures within the Alloy-OM model (Listing 3.2) specify them as singleton signatures. While these constraints are required by the tradeoff space generator (e.g. to not generate multiple tables for a class in the model), they are unneeded for load generation, and thus omitted in the load model. The element attributes in the object model are also declared as fields of the corresponding load signature definition representing relations from the signature to the attribute *type*.

Finally, two sets of constraints are synthesized as *fact* paragraphs in the load model to guarantee both *referential integrity* of generated data as well as *uniqueness* of element identifiers with reference to the set of element instances to be generated. Referential constraints require every value of a particular attribute of an element instance to exist as a value of another attribute in a different element.

Consider the association relationship between Customer and Order classes from our running example (Figure 3.2). The code snippet of Listing 3.4 represents synthesized constraints in the load model for the customer-order association.

```
1 fact {
2   all o1,o2:CustomerOrderAssociation|o1.orderID = o2.orderID and o1.
        customerID = o2.customerID => o1=o2
3 }
4
5 fact {
6   all o:CustomerOrderAssociation| one c:Order| o.orderID = c.orderID
7   all o:CustomerOrderAssociation| one c:Customer| o.customerID = c.
        customerID
8 }
```

Listing 3.4: Part of the load model specification generated for customer-order association.

The expression of lines 1–3 states that if any two elements of type *CustomerOrderAssociation* have the same *orderID* and *customerID*, the elements are identical. This constraint rules out duplicate elements. The fact constraint of lines 5–8 states that for any orderID and customerID fields of a*CustomerOrderAssociation*, there are *Order* and *Customer* instances with the same *orderID* and *customerID*.

Applying the Alloy Analyzer to the derived load model yields the desired load in the form of object model data instances (OM-instances). Figure 3.7 depicts a generated OM-instance, an Alloy solution object. This solution represents two customers with customerID of 64 and 225, the latter a preferred customer with 10 percent discount, along with their orders. From many such solutions we derive an abstract (application-object-model-level, rather than concrete-database-schema-level) load with which to test the performance of many database instances.

Improving the efficiency of the load generator. One of the challenges we faced involved the scalability of this approach to load synthesis. A large number of solutions generated by the Alloy Analyzer were symmetric to previously generated instances, and thus did not contribute usefully to the load being generated. We explored a number of ways to improve efficiency of the load generator. The one that we found worked best is the iterative refinement of the load model by adding constraints that eliminate permutations of the already generated OM-instances. Without this improvement, it took 21 hours for Trademaker-ORM to generate test loads for one of our experiments. Given this approach, the time was reduced to about 2 hours—an order of magnitude speed up in the synthesis of test loads.

3.3.3 Abstract Load Concretization

The next challenge we discuss is to convert abstract load OM-instance objects into concrete SQL queries on a per-database basis. This is the task of specializing abstract load elements to the variant schemas presented by different solutions in the design space. Our *Alloy-to-SQL transformer* handles this task. To create SQL statements for a given database, *Alloy-to-SQL transformer* requires an OR mapping, the abstraction function describing how that concrete database schema implements the abstract object model.

Algorithm 1 outlines this transformation. For brevity, and because it suffices to make our point, this section focuses on *insert* queries. The approach supports the generation of select and update queries as well, which are important, of course, for comprehensive dynamic analysis.

Algorithm 1: Generate SQL Insert Statements

Input: omi: OM-instance, map: OR mapping
Output: A set of SQL insert statements

```
1  for element in omi do
2      T = map.TableAssociat(element);
3      F = T.fields;
4      for field in F do
5          value = getValueFromOMI(field);
6          if value != null then
7              add “field = value" into statements
8          end
9          else
10             if field == “DType" then
11                 value = element.name;
12             end
13             if isForeignKey(field) then
14                 attr = findAttributeFromAssociation(field);
15                 value = getValueFromOMI(attr);
16             end
17             add “field = value" into statements
18         end
19     end
20 end
```

The logic of the algorithm is as follows. Iterate over all elements in a given OM-instance (e.g., classes and associations) whose values can be populated into databases through insert statements. Look up the mapping to determine the table in which the element values should be stored. For each relational field

in the associated table, if the OM-instance contains a value corresponding to that field, insert the value into the field. Otherwise, in the case that the field is a *DType*, insert the name of element into the field. Finally, if the field is a foreignKey, find the associated attribute from a relevant association in the given OM-instance, and insert its value into the field.

Consider the database alternative for our running example, in which we store the customer-order association data into the order table (Figure 3.2b). In that case, the field of *customerID* in the *Order* table is a foreignKey, and its values comes from the associated customerOrderAssociation element.

```
1 INSERT INTO `Customer` (`customerID`,`DTYPE`) VALUES (64,'Customer');
2 INSERT INTO `Customer` (`customerID`,`DTYPE`) VALUES (225,'
      PreferredCustomer');
3 INSERT INTO `Order` (`orderID`,`orderValue`,`customerID`) VALUES
      (184,511,64);
4 INSERT INTO `Order` (`orderID`,`orderValue`,`customerID`) VALUES
      (366,510,225);
```

Listing 3.5: Generated SQL insert statements from OM-instance of Figure 3.7 for implementation mapping of Figure 3.6.

Listings 3.5 represents the set of SQL insert statements generated from the OM-instance of Figure 3.7 according to the mapping of Figure 3.6. The first two generated statements define insert queries to store instances of Customer and PreferredCustomer into the Customer table along with appropriate *DType* values for each one. The next two statements then store instances of Order and CustomerOrderAssociation into the Order table.

3.4 Dynamic Analysis Experiment

As an experimental test of our approach to specification-driven, automated dynamic analysis of non-functional property tradeoffs across design spaces, we have applied the approach to test the validity of the static predictors of database performance. This section summarizes the design and execution of

our experiment, the data we collected, its interpretation, and our results, which include novel findings regarding these metrics.

3.4.1 Static Metrics Suite

The choice of mapping strategy impacts key non-functional system properties. Response time performance, storage space and maintainability are among the set of quality attributes defined by the ISO/IEC 9126-1 standard that are influenced by the choice of OR mappings. To statically measure these attributes in an ORM design space, we use a set of metrics suggested by Holder et al. [25] and Baroni et al. [24]. The metrics are called Table Access for Type Identification (TATI), Number of Corresponding Table (NCT), Number of Corresponding Relational Fields (NCRF), Additional Null Value (ANV), Number of Involved Classes (NIC) and Referential Integrity Metric (RIM). In this work, we focus on three of these metrics for time and space performance. Maintainability is out of scope as we cannot measure it using dynamic analysis technique.

Table Accesses for Type Identification (TATI)

Table Accesses for Type Identification (TATI) is a performance metric for polymorphic queries [25]. According to the definition, given a class C, TATI(C) defines the number of different tables that correspond to C and all its subclasses. Our tools total up TATI values for each class as the overall TATI measure for each solution alternative.

Number of Corresponding Tables (NCT)

Number of Corresponding Tables (NCT) is a performance metric for insert and update queries. This metric specifies the number of tables that contain data necessary to assemble objects of a given class [25]. According to the definition, given a class C, NCT(C) equals to NCT of its direct super class, if C is mapped to the same table as its super class. Otherwise, if C is mapped to its own table,

NCT(C) equals to NCT of its direct super class plus one. Finally, if C is a root class, NCT(C) equals to 1. Our tool computes totaled NCT values over classes as the NCT measure for each solution alternative.

Additional Null Value (ANV)

The Additional Null Value (ANV) metric specifies the storage space for null values when different classes are stored in a common table [25]. According to the definition, given a class C, ANV(C) equals to the number of non-inherited attributes of C multiplied by the number of other classes that are being mapped to the particular table to which C is mapped. Our tools present totals for ANV values over all classes as the ANV measures for each solution alternative.

3.4.2 Static Analysis of Synthesized Designs

To apply these metrics to synthesized solutions, we designed specific Alloy queries. Here we describe one for measuring the TATI metric. The others are evaluated similarly.

$TATI(C) = \#(C.*(\sim parent).\sim tAssociate)$

Here the dot operator denotes a relational join. The Alloy $\sim$ operator represents the transpose operation over a binary relation, which reverses the order of atoms within the relation. Given the *tAssociate* (abstraction) relation that maps tables to their associated elements (i.e. Class or Association) within the object model, its transpose is the relation that maps each element to its associated table within the relational structure. The Alloy * operator represents the reflexive-transitive closure operation of a relation. Accordingly, the expression of "*C.*(∼parent)*" states a set of classes that have the class "C" as their ancestor in their inheritance hierarchy. The query expressions then, by using the Alloy set cardinality operator #, computes the TATI metric.

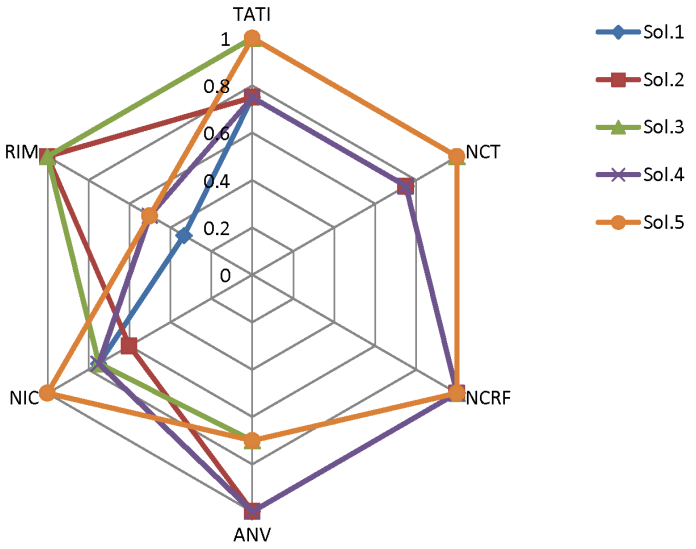

Figure 3.8: Multi-dimensional *quality measures* for pareto-optimal solutions.

Our static metrics suite comprises six such static measures. The vector of these functions defines a 6-dimensional static analysis function applicable to Alloy-synthesized concrete designs (e.g., Figure 3.6). Our tools map this function over all elements of a synthesized design space to produce a tradeoff surface. The spider diagram, shown in Figure 3.8, illustrates one Pareto-optimal point on that surface for our example customer-order system. To display quality measures in one diagram, we normalized the values. Such diagrams can assist in conducting tradeoff analyses by making it easier to visualize and compare alternatives. According to the diagram, if the designer opts for performance, she may decide to use Sol. 5 instead of Sol. 4, as the latter has worse values for the TATI and NCT performance metrics.

Of course none of this theory or machinery is very useful if the metrics themselves are not predictive of the actual non-functional (performance) properties of candidate designs. The rest of this section presents our experiment in automated dynamic analysis, the goal of which was to help us answer this question. In particular, this experiment addresses the three hypotheses introduced at the start of this section.

3.4.3 Subject Systems

We synthesized design spaces and compared static predictions with dynamic results for four subject systems. The first is the object model of an E-commerce system adopted from Lau and Czarnecki [30]. It represents a common architecture for open source and commercial E-commerce systems. It has 15 classes connected by 9 associations with 7 inheritance relationships. The second and third object models are for systems we are developing in our lab. Decider is another system to support design space exploration. Its object model has 10 Classes, 11 Associations, and 5 inheritance relationships. The third object model is for a system, CSOS, a kind of cyber-social operating system meant to help coordinate people and tasks. In scale, it has 14 Classes, 4 Associations, and 6 inheritance relationships. We also analyzed an extended version of our customer-order example.

3.4.4 Planning and Execution

Our experimental procedure involved the synthesis of both design spaces of database alternatives and several abstract loads in a variety of sizes for each subject system. Given the synthesized schemas, we created a database for each alternative. We then populated generated data into databases, and ran concrete queries over those databases. We measured and collected the numbers of concrete queries generated from abstract loads for each database alternative, query execution time, as well as the size of each database.

We used an ordinary PC with an Intel Core i7 3.40 Ghz processor and 6 GB of main memory, with SAT4J as our SAT solver. Database queries were performed on a MySQL database management system (DBMS) installed on a machine equipped with an AMD Opteron 6134 800 Mhz processor and 64GB memory. Data and statistical information are available at http://jazz.cs.virginia.edu:8080/Trademaker/data.

Figure 3.9 summarizes generated solution space for each subject system. There is one row for each system. The columns indicate the total number of solutions, the number of static equivalence classes where equivalence is determined by

Subj. Sys.	Solutions	Eq.Classes	Pareto Sols.
decider	386	154	12
ecommerce	846	360	16
CSOS	278	121	21
Cust-order (ext)	28	14	10

Figure 3.9: Design space sizes for subject systems.

equality of static analysis results, and the number of Pareto-optimal solutions under the given static metrics.

We investigated and compared two different methods for generation of data sets. The first method generated data using our formal synthesis methods. For the second, we hand-developed a load generator for generating large loads that simply respect the constraints in our object models (e.g., referential constraints between elements).

Three data sets were developed for each subject system to support the task of evaluating the static metrics.

Dataset 1. This data set is generated using the Alloy-based data generator, where the maximum bit-width for integers is restricted to 5. This leads to the generation of small data set for our experiments.

Dataset 2. This data set is generated using our Alloy-based data generator. The maximum bit-width for integers is restricted to 10, which leads to the generation a larger data set compared with the former data set.

Dataset 3. As with many formal techniques, the complexity of constraint satisfaction restricts the size of models that can practically be analyzed and synthesized [31, 32]. For experimental purposes, we hand-implemented a more scalable data generator. It does not generate queries directly, but rather replaces the constraint solver for synthesis of abstract loads. Having synthesized larger abstract loads in the form of OM-instances, using the mechanisms already used in the Alloy-based data generator (cf. 3.3.3), the generator then transforms abstract loads into sets of concrete queries targeting diverse implementation alternatives.

Ecommerce	Dataset 1	Dataset 2	Dataset 3
abstract load	862	2,576	164,813
Sol.19 (conc. load)	456	13,698	2,471,700
Sol.121 (conc. load)	320	9,770	1,647,800
Sol.264 (conc. load)	397	12,073	2,142,140
Sol.348 (conc. load)	379	11,395	1,977,360

Figure 3.10: Part of the generated data sets for the ecommerce experiment; the first row shows abstract loads generated for the ecommerce system within each data set; each cell in the other rows corresponds to the size of generated concrete load for the database alternative and data set given on the axes.

		TATI	*NCT*	*ANV*
decider	Dataset1	0.91	0.95	-0.87
	Dataset2	0.98	0.97	-0.77
	Dataset3	0.98	0.97	-0.81
ecommerce	Dataset1	0.9	0.96	-0.91
	Dataset2	0.98	0.97	-0.92
	Dataset3	0.95	0.96	-0.92
CSOS	Dataset1	0.86	0.78	-0.88
	Dataset2	0.7	0.75	-0.69
	Dataset3	0.71	0.80	-0.66
Cust-order (ext)	Dataset1	0.7	0.97	-0.38
	Dataset2	0.74	0.71	-0.76
	Dataset3	0.71	0.91	-0.24

Figure 3.11: Correlation coefficients between the relative order of solution alternatives predicted by static metrics and those observed from actual runtime measures.

Figure 3.10 presents the sizes of generated data sets for some of the solution alternatives for the E-commerce system. The number of concrete queries refined from a common abstract load is different in various solution alternatives, depending on the way that each implementation mapping alternative refines the abstract object model into the concrete representation in relational structure (cf. 3.3.3).

3.4.5 Results for Hypothesis H1 (Order)

To test the predictive power of our static metrics, we compared its predictions against the results of our dynamic analysis. To evaluate our first hypothesis—whether the relative order of implementation alternatives is predicted by static

metrics—we compute Spearman correlation coefficients, an appropriate correlation statistic for order-based consistency analysis. It measures the degree of consistency between two ordinal variables [33]. A correlation of 1 indicates perfect correlation, while 0 indicates no meaningful correlation. Negative numbers indicate negative correlations.

Figure 3.11 summarizes correlation coefficients between static metrics and dynamic measures. The data show reasonably strong but somewhat inconsistent positive correlations between statically predicted and actual run-time performance for TATI (average correlation of 0.84) and NCT (average correlation of 0.89). These metrics appear moderately to strongly predictive of the relative ordering databases run-time performance, at least for the kinds of loads employed in our experiments.

The performance of the ANV predictor varies across the subject systems. ANV predicts well in the E-commerce and Decider experiments and moderately in the CSOS data, but weakly in the customer-order-extended set. Moreover, the results show negative correlation between the ANV metric and database size. As number of null values increases, size decreases. This observation for the ANV metric is in direct contrast to what is predicted. One possible reason is that when the ANV metric increases, the number of tables for the database solution under consideration decreases. Assuming that the database system efficiently stores null values, the database size would reduce.

To further evaluate predictiveness of the static metrics, we considered the case in which designers use each static metric as a two-class classifier. We, thus, measure precision, recall and F-measure as follows:

Precision is the percentage of those alternatives predicted by a given metric as more preferable in terms of a given quality attribute that were also classified as more preferable by the actual analysis: $\frac{TP}{TP+FP}$

Recall is the percentage of alternatives classified more preferable by the actual analysis that were also predicted as more preferable by the a given metric: $\frac{TP}{TP+FN}$

F-measure is the harmonic mean of precision and recall: $\frac{2*Precision*Recall}{Precision+Recall}$

where TP (true positive), FP (false positive), and FN (false negative) represent the number of solution alternatives that are truly predicted as preferable, falsely predicted as preferable, and missed, respectively.

While static metrics output predictions of quality characteristics as natural numbers, actual analysis of query execution performance and required storage space are in terms of Seconds and Bytes, respectively. To classify an alternative as *preferable*, we thus use median for each set of result values as a threshold. We measure evaluation metrics for each subject system with respect to three data sets.

		TATI			NCT			ANV		
		Prec	Recall	F-measure	Prec	Recall	F-measure	Prec	Recall	F-measure
decider	Dataset 1	1	0.75	0.86	0.83	0.83	0.83	0	0	0
	Dataset 2	0.83	0.83	0.83	1	1	1	0.17	0.17	0.17
	Dataset 3	1	1	1	0.83	0.83	0.83	0.17	0.17	0.17
ecommerce	Dataset 1	1	1	1	1.00	1	1	0	0	0
	Dataset 2	1	1	1	1.00	1	1	0	0	0
	Dataset 3	0.9	0.9	0.90	1.00	1	1	0	0	0
CSOS	Dataset 1	0.75	0.82	0.78	0.75	0.82	0.78	0.36	0.33	0.35
	Dataset 2	0.83	1	0.91	0.67	0.80	0.73	0.36	0.33	0.35
	Dataset 3	0.83	1	0.91	0.83	1	0.91	0.36	0.33	0.35
Cust-order (ext)	Dataset 1	1.00	1	1	1.00	1	1	0.43	0.60	0.50
	Dataset 2	0.83	1	0.91	0.67	0.80	0.73	0.57	0.67	0.62
	Dataset 3	0.83	1	0.91	0.83	1	0.91	0.43	0.60	0.50

Figure 3.12: Experimental results of evaluating OR metrics as two-class classifiers.

Figure 3.12 summarizes the results of our experiments to evaluate the accuracy of static metric predictors as two-class classifiers. The average precision, recall and F-measure are depicted in Figure 3.13. The results show the accuracy of the TATI and NCT metrics in classifying implementation alternatives in terms of their run-time performance. The average precision and recall for all four experiments are about 90%, showing a low rate of both false positives and false negatives. The ANV metric, however, achieves the average under 30% in all evaluation metrics.

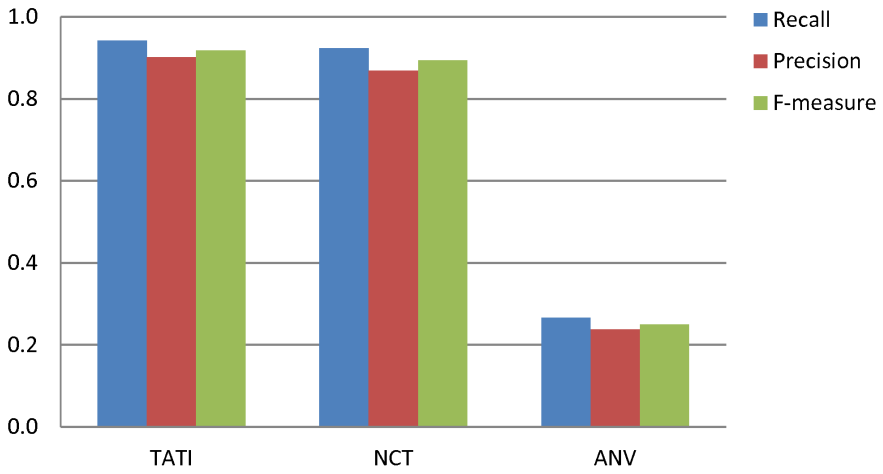

Figure 3.13: Bar plot of the average precision, recall and F-measure for considering static metrics as two-class classifiers.

The experimental data thus suggests that, under the generated abstract loads, the relative order of implementation alternatives predicted by static metrics of TATI and NCT is indicative of their comparative preference in actual runtime performance, but this is not the case for ANV as a static predictor of storage space.

3.4.6 Results for Hypothesis H2 (Magnitudes)

To address the second hypothesis—relative magnitude of static predictions matter—we employ a coefficient of determination denoted R^2, as a metric for how well actual outcomes are predicted by the static metrics. Figure 3.14 plots the results. For brevity, only results from *Dataset 3* are presented; other data sets give similar results.

The performance of the predictors varies widely across systems and predictors. TATI and NCT are predictive of performance for the E-commerce and decider systems, but relatively poor predictors for CSOS. TATI performs poorly in the customer-order-extended data. ANV predicts size in the E-commerce experiment, and moderately in the Decider data, but inconsistently and weakly in the CSOS data and not at all in the customer-order-extended set.

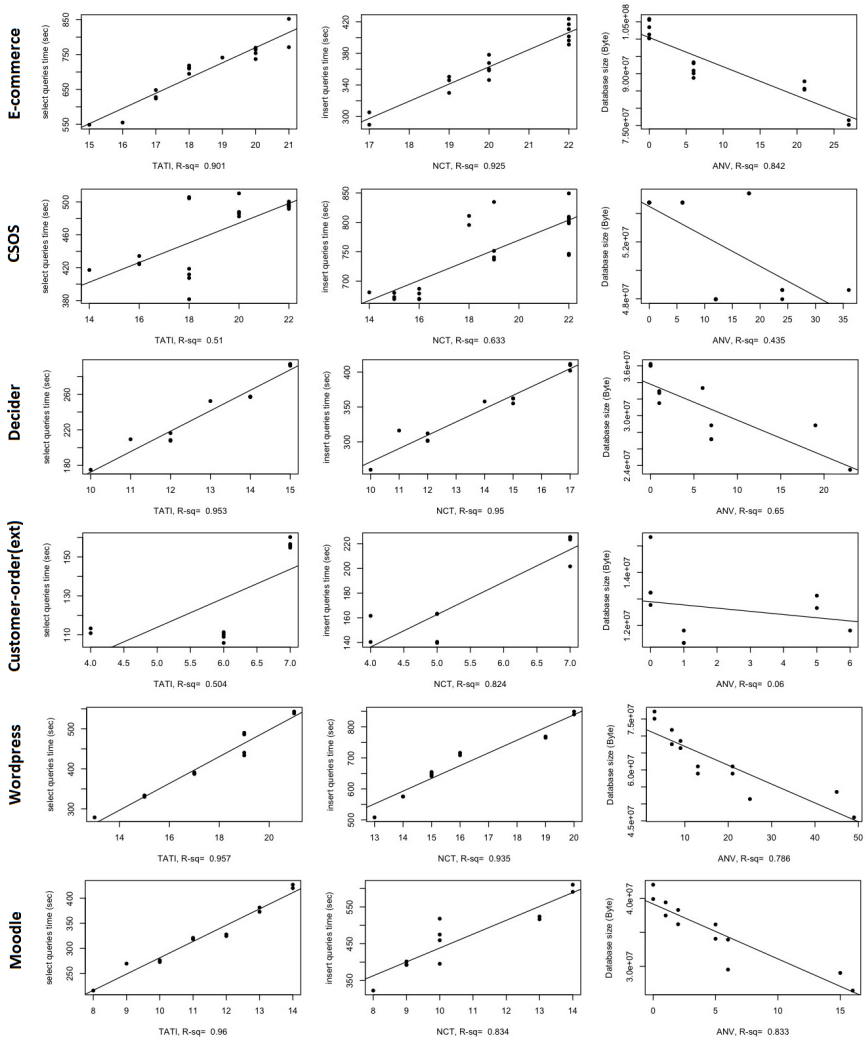

Figure 3.14: Correlation between static metrics and actual run-time measure; rows represent scatter plots of observed values versus predicted values by TATI, NCT and ANV metrics from top to bottom, respectively; R^2 correlation coefficient is shown at the bottom of each plot.

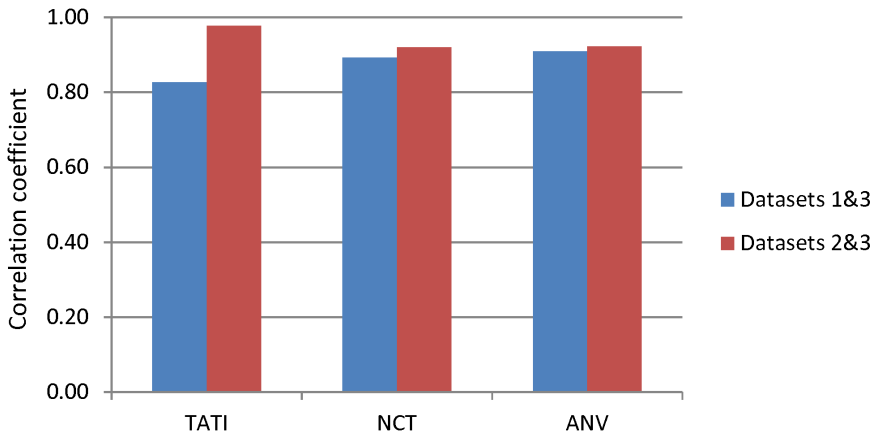

Figure 3.15: Summary of Pearson correlation coefficients between experimental results obtained from smaller data sets and that of the large *Dataset*3.

We interpret this data as suggesting that the relative magnitudes of static metrics for various solution alternatives are not reliably indicative of the relative magnitudes of actual performance, and that ANV is a poor indicator of the storage space. One is advised to use such static metrics with caution. While TATI and NCT metrics predict the relative order of solution alternatives with high confidence, the difference in predicted values of two alternatives is not a good indicator of their actual run-time difference.

3.4.7 Results for H3 (Small vs. Large Loads)

To address the third hypothesis—that small, formally synthesized loads predict the outcomes of much larger loads—we employ the Pearson product-moment correlation statistic. Pearson measures the degree of linear dependence between two variables, not necessarily ordinal, as opposed to the Spearman test. A correlation of 1 represents perfect correlation, and 0, no meaningful correlation.

We summarize correlation coefficients between experimental results obtained from smaller data sets of 1 and 2 and that of the large *Dataset 3* in Figure 3.15. The average Pearson correlation coefficient between *Dataset*3 and the first and second data sets are 88% and 94%, respectively. These data lend support to the proposition that smaller-scale test sets produced by specification-driven

synthesis can provide valid predictions of performance under larger, more realistic loads.

3.5 Evaluation of Trademaker

The overarching problem this work addresses is interesting and important: the need for improved science to support decision making in complex and poorly understood tradeoff spaces, particularly involving tradeoffs among non-functional properties, also sometimes called *ilities*. We need languages in which to specify design spaces, techniques for synthesizing and analyzing design spaces, mechanisms for mapping static and dynamic analysis functions across design spaces, techniques for validating such metrics, and tools that enable engineers use the science to improve real engineering practice. We also need measures for *ilities* that are important but hard to measure today: for evolvability, some dependability properties, affordability of construction, and more.

This work contributes some new results to the science and engineering of tradeoff analysis of non-functional properties. It suggests the possibility of useful formal languages for specifying design spaces in support of formal synthesis of both designs and comparative analysis loads. We showed that specialization of common loads is enabled by access to abstraction functions from concrete to abstract designs, which can be embedded in the results of design synthesis. Concretization functions proved useful not only for scale-limited, formally synthesized loads, but for concretizing abstract loads produced by other means. We also presented an experiment using our tool to test the validity of static predictors of database performance based on published but not validated metrics. Two of the metrics appear to produce meaningful signals, while the third appears not useful. The data also indicate a need for caution in relying on the static metrics. Their predictive power, even in the “good" cases, varied across application models. That said, we can now provide automated dynamic analysis as a fall-back. We are integrating support for invoking such automated analysis into our Trademaker-ORM tool. Trademaker-ORM itself has real potential utility for object-relational

mapping and partial application synthesis; but its greater significance is as a demonstration of our research results and a testbed for further research on formal tradespace modeling and analysis.

There are of course limitations in our approach and in this work. We mention those most relevant to a proper evaluation of this effort. First, the static metrics we evaluated *sum* the values of published metrics over the elements of each design alternative. We thus extended the original metrics and our statistical results should technically be read as pertaining to these extensions of the original measures.

Second, while our synthesis mechanisms are implemented and working, our infrastructure for running synthesized concrete loads against synthesized designs still relies on some manual processing. Our statistical data were thus derived by dynamic analyses of certain *subsets* of our synthesized designs. We selected the subsets deemed Pareto-optimal by the static metrics. As our infrastructure matures, we will conduct whole-space dynamic analyses, which we expect to produce results consistent the basic result presented here. We are on a path to support automated whole-space dynamic analysis through Trademaker-ORM. The work reported in this work did nevertheless involve the synthesis and dynamic analysis of over 300 database alternatives.

Third, our experiments to date tested our hypotheses for “random" loads of varying sizes. Real applications will generally produce non-random loads. Whether the static metrics we tested are predictive for large, real applications remains unclear. On the other hand, we offer dynamic analysis at scale as an alternative. We envision a future in which some systems run many design variants in parallel, perhaps with small but representative loads abstracted from real loads on live systems, to detect conditions in which dynamic switching to new implementation strategies should be considered.

Finally, there is the issue of scalability. Using Alloy as a constraint solver entails scalability constraints. We can handle object models with tens of classes. Industrial databases often involve thousands of classes. It is unlikely that our current implementation technology will work at that scale. For now, it does

have real potential as an aid to smaller-scale system development. That we can present an object model for a realistic web service, synthesize a broad space of ORM strategies, select one based on tradeoff analysis, automatically obtain an SQL-database setup script, provide it Java EE, and have much of an enterprise-type application up and running with little effort is significant.

3.6 Astronaut: An Automated tradeoff Analysis Framework

Our previous experiments in the ORM domain were conducted manually. For all synthesized designs, we manually execute command line tools to evaluate time and space performance. However, a large model could lead to tens of thousands designs. Manually evaluating that many designs are not possible, which in turn required a new apparatus. In this section, we present the design and implementation of *Astronaut*, a framework for automated tradespace analysis. It fully automates the end-to-end conversion of system specifications into dynamically profiled tradespaces, i.e., exhaustive *dynamic* relational-logic-based tradespace analysis. Its design and evaluation is the second contribution of this work. Our previous work of Trademaker dynamically profiled only a few hundred designs, at a cost of months of manual effort, to enable statistical testing of the predictive accuracy of certain *static* database performance predictors (which operation directly on static design models of SQL schemas).

Astronaut in turn implements an abstract model for TA, presented informally in our previous work [34]. However, we never formalized or mechanically checked our model. The main contribution of this work is a formal specification, written in the logic of Coq and checked for consistency, from which we extracted the Scala implementation of Astronaut. The Coq specification is polymorphic in types that encode the syntax and semantics of a given domain-specific specification language, the solver that will be used to enumerate designs, the set of property estimation or measurement procedures used to profile designs, etc. These parameters are extracted to Scala as abstract classes that are overridden to instantiate Astronaut

for a given domain. This work involves re-engineering ORM-specific code of Trademaker into sub-classes that we then plugged into Astronaut to instantiate it as an ORM tool. We present our Coq specification as a straightforward, abstract, formalized, generalized, and evolvable theory of dynamic TA, from which code can be extracted automatically, and which we have tested for utility by using it to create Astronaut.

A comment on scalability is in order. RLTA, thus Astronaut, involves exhaustive enumeration of models of bounded relational logic models. This is a *#P-complete* problem: harder than *NP-complete* and equivalent to counting the number of satisfying solutions to a SAT problem. It is intractable in general, and will not scale to large, complex systems. Yet model checking tools have clearly demonstrated the potential value in exploring practical uses of solutions to theoretically intractable problems. RLTA is no panacea. It might be useful. Here we remain satisfied to explore its potential utility for problems at the scale of individual modules, such as database schemas for ordinary web applications. While the modular architecture of Astronaut supports variation in logics and solvers thus enumeration strategies, we leave the exploration of such variations on our theme to future work.

3.6.1 Astronaut Design

This section presents an overview of our approach for automated tradeoff analysis, and describes how it leverages advances in several areas of computer science and engineering.

Constructive Logic & Certified Programming with Dependent Types

First, we use the expressiveness of dependent type theory in modern constructive logic proof assistants to produce precise, yet computationally effective, theoretical framework for tradeoff analysis. We present an algebraic theory of tradeoff analysis tools structured as a hierarchy of Coq [35] typeclasses, in a style similar

to that being used by mathematicians [36,37] to formalize hierarchies of abstract algebraic structures (e.g., groups, fields, topological spaces).

From this theoretical framework, we use Coq's extraction facility to derive a certified [38] implementation of a general-purpose syntesis framework in Scala. It is then specialized and extended with user-defined, domain-specific types and functions, subject to the specified laws, to create domain-specific analysis tools. The framework provides implementations of functions common to all instances and expresses laws that all instances must obey.

Formal Synthesis from Specifications

Our framework is meant to be specialized using any types of software synthesis techniques. In this work, we specialize it in the context of database schema designs for object-oriented applications. We use a relational logic model finder to exhaustively synthesize relational database schemas as well as test loads for dynamic analysis of performance from specifications of object-oriented data models [23,34]. With this tool, we are now able to fully replicate the largely manual analysis of synthesized database schemas reported in our earlier work [34].

Scalable Big Data Analytics

We use big data analytics, particularly map-reduce [39], to reduce analysis runtimes. While synthesizing spaces of design solutions from specifications may not always be easily parallelized, applying property analysis functions to dynamically measure each design solution in multiple dimensions of performance is. In fact, applying each measurement function to each solution has a natural map-reduce structure. The use of scalable map-reduce technology can benefit many instances of our framework, so it is practical to support it as a common middleware plug-in. In the following sections, we describe the details of our approach.

3.6.2 A Constructive Logic Based Framework

This section presents our framework of tradeoff analysis tools as a hierarchy of typeclasses in Coq. We identified that such a framework must support: 1) different types of formal specification inputs, which we take to be incomplete in general regarding all properties of interest; 2) different kinds of synthesizers to solve the given specification to produce implementations; 3) different kinds of test loads for dynamic analysis, and specialization of common loads to the interfaces exposed by particular implementations; 4) different types of system property measurement functions; 5) different types of analysis results; and 6) different kinds of filters to select optimal designs based on analysis results.

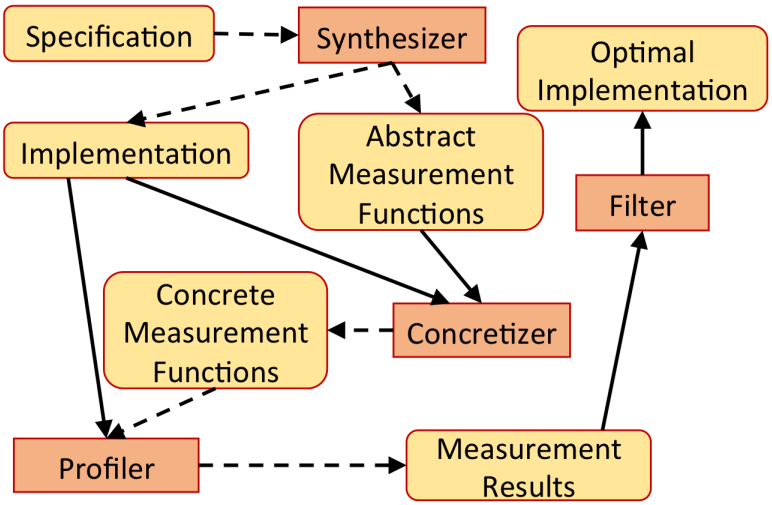

Figure 3.16: Framework Data Flow

We have exploited the idea of using typeclasses to capture general families of mathematical structures by having our typeclass-based specification define the *family* of Astronaut instances, including dimensions of variation, shared operations, and invariants common to all family members. Typeclasses in Coq are useful for specifying interrelated *families* of mathematical structures, including carrier sets, operations, and invariants, as well as for specifying instances of these structures. An example of interrelated families of structure from abstract algebra include monoids, groups, and fields (each of which enriches the previous one).

The structure of our framework is shown in more detail in Figure 3.16. It takes user defined specifications as inputs. The synthesizer then synthesizes a set of implementation and associated test loads. Next, the measurement function dynamically analyzes each implementation with associated test loads to get a set of measurement results for each solution point. At last, the filter picks the set of (Pareto-)optimal solutions based on the measurement results.

Astronaut Typeclass

Figure 3.6 shows Astronaut, which is the most abstract and least structured typeclass in our hierarchy. It expresses the functionality of a broad family of tradespace analysis tools at a high level of abstraction. The code fragments in Listing 1 presents the hierarchy of abstract algebraic structures that we formalized as Coq typeclasses. Each typeclass (Set in the code) characterizes a class of possible instances. Each class represents a type, values of the type, operations on the type, and laws constraining the behaviors of the operations. It has four types: (1) type of input specification (e.g., a UML class diagram); (2) type of derived implementation (e.g., SQL schema); (3) type of a set of property measurement functions (e.g., space performance measurement); and (4) type of measurement results (e.g., database size in megabytes). Values have to be provided when instantiating Astronaut typeclass.

```
1  Class Astronaut := {
2    SpecType: Set
3  ; ImplType: Set
4  ; MeasureFuncSetType: Set
5  ; MeasureResultSetType: Set
6  ; synthesize : SpecType ->
7      list(ImplType × MeasurFuncSetType)
8  ; mapReduce: list (ImplType ×
9        MeasureFunctSetType) ->
10     list (ImplType × MeasureResultSetType)
11 ; astronaut (spec: SpecType):
12     list (ImplType × MeasureResultSetType) :=
13     map mapReduce (synthesize spec)
14 }.
```

Listing 3.6: Astronaut Typeclass Coq specification, which defines the most abstracted framework structure. It takes a specification as input, and outputs a set of implementations and measurement results.

The *synthesize* component is a function that maps a specification to lists of $< Implementation, MeasurementFunction >$ pairs. The measurement functions provide performance comparisons of variant implementations. The *mapReduce* component is a function that takes a list of $< Implementation, MeasurementFunction >$ pairs, and returns a list of $< Implementation, MeasurementResult >$ pairs. The *astronaut* function calls the *map* function to map the measurement functions over the output of the synthesize function (a list of $< Implementation, MeasurementFucntion >$ pairs). Implementations of these functions are required when the typeclass is instantiated.

Tradespace Typeclass

The Astronaut typeclass[2] (Listing 1) captures a very general notion of tradespace analysis, and nicely illustrates some of the aspects of our approach, but it provides too few details for implementing a tradespace analysis tool. We introduce a new tradespace typeclass to enrich the Astronaut typeclass to provide a finer-grained implementation architecture for tradespace analyzers.

Listing 3.7 partially presents Tradespace typeclass Coq specification. The first two lines state that the *Tradespace* class extends (is coercible to) the Astronaut and ParetoFront typeclasses. The latter provides structure for computing Pareto fronts of sets of vector-valued objects. The following four components (lines 5–8) provide for parameterization of typeclass instances with respect to the key additional types of the implementation framework. Following the declarations of these type-valued parameters, lines 10–21 specify the signatures of the mapping functions required to instantiate the Tradespace typeclass.

The diagram shown in Figure 3.17 graphically depicts the structure of the Tradespace typeclass. The concretization function (cFunction) maps the formal specification to a set of formal representations of implementations (FmImplType). The function abstraction (aFunction) explains how each implementation represents and satisfies its specification. The load function (lFunction) maps the same

[2]Research artifacts and the complete model for Astronaut, including all specifications, are available at http://chongtang.github.io/Astronaut/

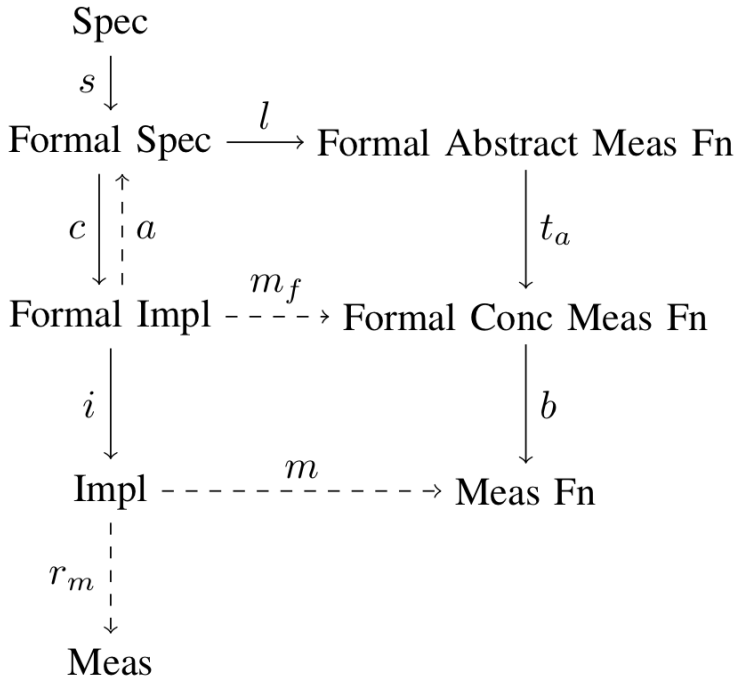

Figure 3.17: Tradespace typeclass model

formal specification to a set of abstract measurement functions (FmAbsMeasureFuncSetType) that will be used to produce concrete measurement function (FmConcMeasureFuncSetType) to measure the properties of implementations. The function tFunction is a conceptual function serves our explanation here and doesn't require implementation. It specializes the abstract measurement functions to the particular interfaces exposed by variant implementations. This function uses the abstraction function aFunction to do its work. The result of this process is another conceptual function/relation, as indicated by m_f in Figure 3.17, that associates a vector of implementation specific measurement function(s) to each implementation.

The function sFunction translates a user given specification to an internal formal specification that can be solved by cFunction. The function iFunction parses the formal/internal representation of implementations to usable forms: e.g., Alloy solutions to SQL schemas. The function bFunction similarly parses formal/internal representation of measurement functions to useful forms: e.g., to objects that run actual SQL scripts against actual databases. The final tradespace analysis result is the relation r_m in Figure 3.17 that associates implementations with

their corresponding property measurement vectors.

The final three components (lines 23–30) specify laws that the other components of the typeclass must follow. In a nutshell, these laws state that the abstraction function, a, must invert the concretization function, c, and that the two paths from specification to measurements must yield the same results. Constructing an instance of the typeclass requires proofs of these propositions for the given function and data type parameter values.

```
1  Class Tradespace := {
2    tm_Astronaut :> Astronaut
3  ; tm_ParetoFront :> ParetoFront
4  (* Internal, Formal-Spec-based types *)
5  ; FmSpecType: Set
6  ; FmImplType: Set
7  ; FmAbsMeasureFuncSetType: Set
8  ; FmConcMeasureFuncSetType: Set
9  (* Internal, Formal-Spec-based functions *)
10 ; cFunction: FmSpecType -> list FmImplType
11 ; aFunction: FmImplType -> FmSpecType
12 ; lFunction: FmSpecType ->
13                FmAbsMeasureFuncSetType
14 ; tFunction: FmAbsMeasureFuncSetType ->
15                list ImplType ->
16                list FmConcMeasureFuncSetType
17 (* map to and from Formal-Spec-based form *)
18 ; sFunction: SpecType -> FmSpecType
19 ; iFunction: FmImplType -> ImplType
20 ; bFunction: FmConcMeasureFuncSetType ->
21                MeasureFuncSetType
22 (* Laws *)
23 ; aInvertsC: forall (spec: FmSpecType)
24                (fImpl: FmImplType),
25                In fImpl (cFunction spec) ->
26                (spec = aFunction fImpl)
27 (* See code repo, omission on purpose. *)
28 ; implLineLaw: ...
29 (* See code repo, omission on purpose. *)
30 ; testLoadsLineLaw: ...
31 }.
```

Listing 3.7: Tradespace Typeclass Coq specification. It captures the internal structure of our framework.

3.6.3 Framework Instantiation

Framework users need to provide domain-specific types for the nodes in Figure 3.17 and domain-specific function implementations for the solid arcs. The other dashed-line mappings are implicit or automatically computed. This section describes how we instantiate our framework to create an automation tool in the context of the ORM domain.

To instantiate an automated tool based on the framework for ORM tradespace analysis, we defined a *DBFormalSpec* class as an actual parameter for the *FormalSpec* slot in this architecture (cf. Fig. 3.17). Concretely, it is a wrapper around a file containing Alloy specification of an object model. Similarly, we created a *DBImplementation* class that wraps a file containing a MySQL schema definition, as a parameter for the *Impl* in Figure 3.17. Our implementation of the function c realizes our Alloy-based approach to synthesize database schemas from object models. Our other ORM-specific values are similar in their structure: classes (Scala types) wrap representation details and function implementations such that they encapsulate details of computations of the various mappings required to implement our tradespace analysis approach. We also implemented *mapReduce* function in Astronaut typeclass using Spark library for Scala. When the tool actually runs, it sends measurement jobs to a pre-configured Spark cluster.

We then provided several parameter values to instantiate the framework. The components include: 1) An Alloy-based ORM domain specification language; 2) an Alloy-based synthesizer to generate candidate designs and test loads (INSERT and SELECT SQL scripts) to be executed to dynamically analyze database designs; 3) two kinds of measurement functions (*Time* and *Space* measurement functions) to measure time and space tradeoffs; 4) a triple of population time, retrieval time, and space consumption as a group of analysis results; and 5) a Pareto-optimal filter to filter out non-Pareto-optimal designs.

Our instance for ORM analysis of this framework is produced with the following parameters.

- SpecType: Object models in formal Alloy-based notation
- ImplType: SQL schema
- MeasureFuncSetType: an instrumented test harness for profiled execution of synthesized SQL scripts
- MeasureResultSetType: a tuple of time and space performance measures from instrumented benchmark execution

- synthesize: given an object model, produce a list of SQL schema and SQL script pairs (INSERT and SELECT statements)
- mapReduce: run a profiled SQL script on a database with the given schema in map-reduce style
- sFunction: An identity function (since the Specification Type in our case is already a formal object model)
- cFunction: Alloy based analyzer that synthesizes an object model to a set of database designs in the form of database creation scripts
- aFunction: A function that returns the mapping information in each database design
- lFunction: A generator that creates formal abstract measurement functions from an object model
- tFunction: A concrete formal measurement function specializer that mapping a formal abstract measurement function to a set of concrete measurement functions based on mapping information of each designs
- iFunction: A parser that parses the database schema embedded in Alloy solution (XML files) to SQL database initial script
- bFunction: An identity function (the formal concrete measurement function in our case is already INSERT and SELECT SQL statements)

3.6.4 Parallelization Reasoning

Even simple specifications could map into a vast number of solutions in the design space. The combination of ample computing capacity and our ability to synthesize immense solution spaces for given specifications suggests that we measure properties and tradeoffs of actual running systems in the spirit of what Cadar et al. called multiplicity computing [26], with the goal of producing new speed-ups in tradeoff analysis of real-world software systems.

Recall that tradeoff analysis using our approach consists of three steps: (1) Solution space and abstract measurement functions are synthesized from formal specifications. (2) The synthesized implementations and abstract measurement functions are transformed into concrete formats, e.g., here formal models are

Algorithm 2: Profiling a given design solution in a worker node.

Input: i: Formal Impl, // **design to be analyzed**
f_a: AbstractionFn, // **abstraction function**
M_a: List<AbstractMeasFn>
// **abstract measurement functions**
Output: p: Result // **profiling evaluation results**

```
1:  M_c = newList() // concrete measurement functions
2:  for m_a ∈ M_a do
3:     m_c ← generateConcreteMeasure(m_a, f_a, f)
4:     M_c.add(m_c)
5:  end for
6:  profResults ← newList()
7:  for m_c in M_c do
8:     for i in [1 : 3] do
9:        result ← runMeasureFunction(i, m_c)
10:       profResults.add(result)
11:    end for
12: end for
13: p ← average(profResults)
14: return p
```

concretized into database schemas and corresponding test loads. (3) Synthesized solution spaces are dynamically analyzed by profiling performance of each solution under its corresponding structure-specific measurement function. In this section, we focus on steps 2 and 3 that can significantly benefit from parallelizing the process.

Similar to a conventional cluster computing paradigm, Astronaut's approach for parallelization consists of a large number of worker machines managed by a master node referred to as the cluster manager. Once Astronaut's cluster manager deploys a synthesized solution to a worker machine, it runs to profile the performance of the given design solution in parallel with other worker machines.

Algorithm 1 outlines the process of profiling a given design solution as realized in a worker node. It takes as input (1) a design solution to be analyzed, i, (2) its associated abstraction function, f_a, that explains how the given design solution instantiates its abstract specification, and (3) a vector of measurement functions, which are essentially abstract test loads automatically generated from the specification and need to be concretized for the given design solution.

The logic of the algorithm is as follows. For each abstract measurement function, $m_a \in M_a$, generate concrete measurement functions, or test loads, that meet the particular structure of the given design solution, i. In doing so, $generateConcreteMeasure$ relies on the abstraction function that relates the design solution back to the abstract specification from which input measurement function, or abstract test load, is derived, and that from these abstraction functions it derives functions for concretizing the given abstract loads. After the measurement functions are generated for the given design, run them to profile its performance.

3.7 Empirical Study

This section describes how we use the tool instantiated from the framework to carry out database schema design and tradespace analysis. More importantly, we discuss the resulting design of synthesis + tradespace analysis and the comparison between the resulting design with two other commonly used tools that built in Django and Ruby on Rails. Previously, we only studied how synthesized schemas perform on MySQL DBMS due to the limit of carrying out experiments manually. In this empirical study, we also aim to investigate how they perform on different DBMSs. Specifically, we study how they perform on PostgreSQL, yet another popular DBMS. We conduct an experimental evaluation that addresses the following research questions:

- **RQ1.** Does Astronaut tradespace analysis enable production of Pareto-optimal designs that are entirely overlooked by widely-used, industrial ORM tools?
- **RQ2.** How well does Astronaut perform? What is the performance improvement achieved by Astronaut's designs compared to those produced by industrial frameworks?
- **RQ3.** What is Astronaut's overhead in conducting tradespace analysis?

- **RQ4.** In the ORM domain, how do the synthesized schemas perform across different database management systems?

This section summarizes the design and execution of our experiment, the data we collected, its interpretation, and our results.

Experimental Objects

Our experimental subjects are selected from different sources and of a variety of different domains, ranging from our research lab projects to applications adopted from the database literature and open- source software communities.

Besides the four subject systems studied in previous experiment, we add three more systems in this empirical study. The fourth object model is the object model of a documents sharing application called Flagship Docs built with Ruby on Rails at Rensselaer Polytech [40]. It has 6 classes connected by 8 associations with no inheritance relationships. We also analyzed an extended version of our customer-order example [34].

The selected experimental subjects are representatives of large classes of useful applications at a scale matched to the state-of-the-art synthesis techniques. Their databases have multiple tables, and several relationships among these tables. There are multiple ownership relationships among these tables, which induce many possible choices of object-model to relational schema mappings.

Experimental Setup and Data Collection

We followed several steps to address the research questions. First, we wrote database specifications using Django, Rails, and the object-modelling language developed by Bagheri et al. [34], for each one of the five systems. We chose to focus on time and space performance of *CREATE* and *READ* transactions for dynamic analysis of target database performance. We wrote instrumentation code to return the desired time and space consumption data into the map-reduce computation. We used the *mysql* client command to execute synthesized test

loads, which our tool parses from internal Alloy representations into MySQL scripts. The first sets of statements were database structure creation scripts. Then *insert* scripts were executed to populate each database with the test data generated in previous stages. The experimental data collected includes the time to run these commands as well as the space consumed by the databases. After populating the databases with test data, we executed *READ* transactions, which are essentially collections of *select* commands. We ran each script three times to rule out other uncontrolled factors that might influence the data we collected. The evaluation results are a list of triples: insertion time, retrieval time, and space consumption. The final result reported in this work is the average time and space performance.

Astronaut vs. Industrial Platforms

Figure 3.18 depicts tradeoff analysis plots produced by Astronaut. It projects the results into 2-D tradeoff plots: Each plot presents tradeoff information in two of the three dimensions for one subject system. Each dot in the plots represents the analysis result of one schema. The hollow triangles that connected by a line are the Pareto-optimal designs in corresponding tradeoff space, exhibiting Pareto-optimal solutions in each plot. We also find out the two schemas produced by *Rails* and *Django* from the synthesized schemas, by looking up the structure in SQL scripts. We then locate the analysis results of these two schemas, and mark them in the 2-D plots with different color and shapes. The hollow star is the design created by Django. The hollow diamond depicts the design generated by Rails. One can easily distinguish the schemas from the other synthesized schemas, indicating where the *Rails* and *Django* generated schemas are fall in the tradespace in terms of the triple measures.

Consider the CSOS Insert-Space tradeoff plot (Row 2 and Column 2) as an example. It shows the tradeoff information of insertion time (in seconds) and space consumption (in megabytes). The two triangle depicts the Pareto front in these two dimensions. It could be just one triangle if there were only one design on the frontier. We can observe that although the schemas created by these

tools are not same, their performances outcomes are close together. The most interesting point is that both of the generated schemas by these tools are far from the Pareto fronts. The same phenomena appear in other plots as well.

The experimental data thus suggests that Astronaut tradespace analysis enables production of Pareto-optimal designs that are entirely overlooked by widely-used, industrial ORM tools.

Improvements in Practice

Table 3.1 presents the performance improvement in the insertion time by comparing a Pareto-optimal schema produced by Astronaut to the performance of databases created by Rails and Django ORM systems. The first line are the five experimental subjects. The second line is the insertion time consumed by the Astronaut's designs. The third line is insertion time consumed by the *Rails* designs, and the forth line is insertion time consumed by *Django* designs. Finally, the last line represents the average performance improvement over *Rails* and *Django* designs across subject systems. Table 3.2 and table 3.3 similarly tabularize the performance improvement in the retrieval time and space consumption.

From the experimental results, we can observe that the Pareto-optimal designs revealed by our analysis generally have much better performance in all dimensions: insertion times, retrieval times, and space consumption. Based on these analysis results, we conclude that our synthesis technique tends to perform better than commonly used tools, particularly, *Rails* and *Django*. Our synthesized designs have better performance, in three models: *CustomerOrder*, *CSOS*, *ECommerce*, and *Decider*, in all three dimensions. For the *Flagship Docs* model, *Rails* and *Django* produced designs equal to ours in performance. We have ascertained that the reason is that the *Flagship Docs* data model has no inheritance relationship, so all of the objects in the object model are mapped as an independent tables in synthesized database schemas.

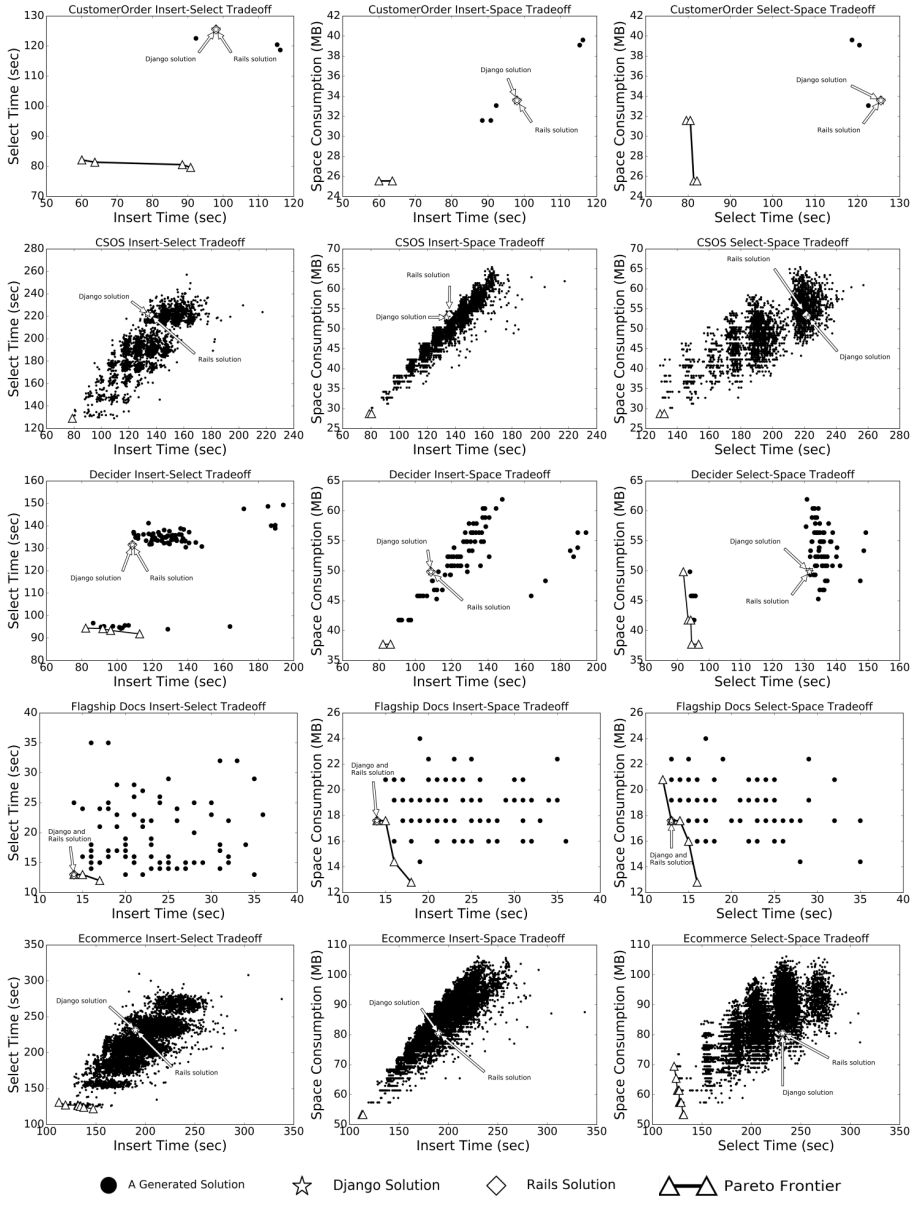

Figure 3.18: tradeoff analysis results; columns represent tradeoff diagrams across systems in two dimensions of Insert-Select, Insert-Space and Select-Space from left to right, respectively; each black dot represents performance of a synthesized database schema, and the star and diamond entries plot the results of schema generated by Django and Ruby

Subject	CO	CSOS	EComm	Decider	Flagship
Astronaut	60.01	78.89	112.64	82.24	13.23
Rails	97.99	133.61	189.93	108.62	13.23
Django	97.99	135.41	189.93	108.62	13.23
Average Improvement	63.3%	70.5%	68.6%	32.1%	0%

Table 3.1: Insertion Performance (Seconds)

Subject	CO	CSOS	EComm	Decider	Flagship
Astronaut	82.12	129.10	131.09	94.46	14.03
Rails	125.54	222.25	231.66	131.57	14.03
Django	125.54	221.25	231.66	131.57	14.03
Average Improvement	52.8%	71.8%	76.7%	39.3%	0%

Table 3.2: Retrieval Performance (Seconds)

Performance and Timing

The final evaluation criteria are the performance benchmarks of tradespace analysis. To carry out the experiments, we set up a 16-node Spark cluster as a back-end analysis platform. Each node has a 2-core AMD Opteron(tm) 242 CPU with kernel clock frequency of 1.5GHz, and 3 gigabytes of memory in total, where 1.9 gigabytes of memory is allocated to Spark. The worker nodes have MySQL server and client tools installed.

Table 3.4 shows the analysis time taken to produce tradespaces across subject systems. The first line are the five experimental subjects. The second line represents the size of tradespace for each subject system. Finally, the last line shows the time required to automatically conduct the tradeoff analysis.

Subject	CO	CSOS	EComm	Decider	Flagship
Astronaut	25.55	28.72	53.34	37.73	17.64
Rails	33.58	52.86	80.56	49.83	17.64
Django	33.58	53.36	80.56	49.83	17.64
Average Improvement	31.4%	85.0%	51%	32.1%	0%

Table 3.3: Space Consumption (MB)

The experimental results confirm that Astronaut approach is effective in systematic dynamic tradespace analysis. In this particular case, the generated Pareto-optimal solutions provide crucial performance improvements to both systems designers and their end-users. Such analysis is not possible with state-of-the-practice tools (e.g., Django and Rails) that produce a single point solution. The results are fascinating, especially when compared with the state-of-the-art technique that requires more than two person months of efforts, during which they were able to analyze only a small sample of representative designs [34]: 14 for CustomerOrder, 121 for CSOS, 154 designs for the Decider, and 360 for the ECommerce model. In fact, Astronaut automates the complete dynamic analysis of exhaustively enumerated design spaces. Clearly, Astronaut does vastly reduce the time required for exhaustive dynamic tradeoff analysis. We expect that performance of Astronaut improves by leveraging more capable computing resources, such as Amazon Web Services, for its parallel reasoning.

Subject	CO	CSOS	Decider	Ecomm	Flagship
Solutions	28	3872	144	14400	256
Time	4 m	5.2 h	15 m	17.8 h	17 m

Table 3.4: Analysis time to produce tradespaces across subject systems

Performance Consistency Cross Different DBMSs

Previously, we studied how the synthesized schemas perform on MySQL and compared their time and space performance with Django and Rails. A question left unanswered is that whether the results are consistent across different DBMSs. Therefore, in this section, we explore how they perform on PostgreSQL.

In this section, we studied two more subject systems from real world besides those considered previously. The first extra system is called *MoodleGrade*, it is the grade sub-module of *Moodle* learning management system. It has eight classes, two associations, and four inheritance relationships. The second extra model is called *Wordpress*, which is the database model of the blog platform Wordpress. It has 13 classes connected by five associations and eight inheritance relationships.

Figure 3.19 shows tradeoff analysis results of studied subject systems. Each row presents the result of one system. From left to right, each column presents insert-select tradeoff, insert-space tradeoff, and select-space tradeoff. Compare to Figure 3.18, which shows the analysis results on MySQL, these results are consistent with them. These results show that our tradespace analysis approach can find Pareto-optimal schemas which work consistently well in MySQL and PostgreSQL.

3.8 Related Work

Our tradespace analysis approach spans and leverages techniques from three research areas: software synthesis, tradeoff analysis, and search-based software engineering.

Software synthesis. There is a large body of research on synthesis techniques. Dang [3] provided a tool for embedded software synthesis. Andersen [41] provided a framework for mathematical problems synthesis. Le [4] provided a framework for data extraction from different kind of sources, like text file, web pages, and spreadsheets. Gupta [16] provided a high-level synthesis framework to transfer a behavioral description in ANSI-C to register-transfer level VHDL with parallel compiler transformation technique. Assayed [5] provided a synthesizer for multi-threads software on multi-processor architectures. Our work in this chapter is aimed to provide a general framework for both tradeoff synthesis and analysis. The plugin-able feature enables the support of arbitrary DSLs and different forms of final implementations. The users can create their own instance by just providing necessary components.

Sketeching [7] similarly is a synthesis technique in which programmers partially define the control structure of the program with holes, leaving the details unspecified. This technique uses an unoptimized program as correctness specification. Given these partial programs along with correctness specification as inputs, a synthesizer – developed upon a SAT-based constraint solver – is then used to complete the low-level details to complete the sketch by ensuring that no asser-

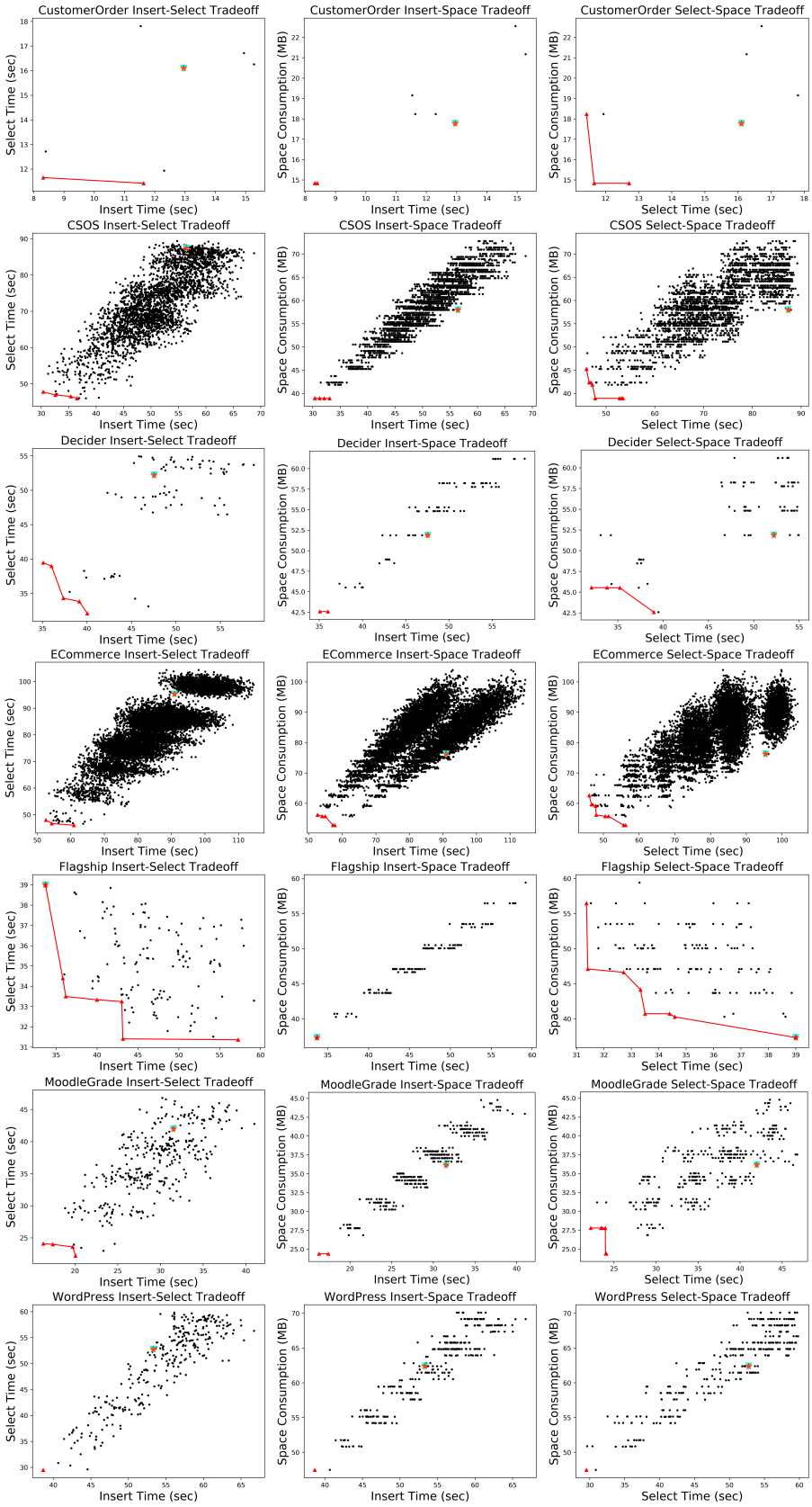

Figure 3.19: Tradeoff analysis results of subject systems on PostgreSQL

tions are violated for any inputs. This work shares with ours the common insight on both using incomplete specifications and synthesis based on constraint solving. However, it does not perform a tradespace analysis over possible completions of a sketch. The synthesis refers to the concrete example of correct or incorrect behavior to prune the design space, and finally narrow down the design space to one implementation that satisfying the given specification.

Different from all these techniques, Astronaut tackles the automated tradeoff space analysis through synthesizing spaces of design alternatives and common measurement functions over such spaces. It thus relieves the tedium and errors associated with their manual development.

Formal derivation of database implementations. A number of researchers have proposed formal approaches for deriving database-centric implementations from high-level specifications [42, 43]. Alchemy refines Alloy specifications into PLT Scheme implementations with a special focus on persistent databases [42]. Along the same line, Cunha and Pacheco proposed an approach that translates a subset of Alloy into the corresponding relational database operations [43]. Both Alchemy and Cunha and Pacheco's approach refine the specification into a single implementation, whereas Trademaker generates *spaces* of possible database design alternatives. While these research efforts share with ours the emphasis on using formal methods, our work differs fundamentally in its emphasis on the generation of *spaces* of implementation alternatives, not just point solutions.

Object-relational mapping. A large body of work has focused on object-relational mapping approaches to the object-relational impedance mismatch problem [27–29, 44]. Philippi [29] categorized the mapping strategies in a set of pre-defined quality tradeoff levels, which are used to develop a model driven approach for the generation of OR mappings. This work similar to many other work we studied derives a single design solution from input specifications. Moreover, they did not apply static metrics, nor dynamic analysis to measure the effectiveness of design alternatives. Our technique is inspired in part by the work of Cabibbo and Carosi [27], discussed more complex mapping strategies for inheritance hierarchies, in which various strategies can be applied independently

to parts of a multi-level hierarchy. Our approach is novel in having formalized ORM strategies previously informally described in some of these research efforts, thereby enabling automatic generation of OR mappings for each application object model.

Drago et al. [45] considered OR mapping strategies as a variation points in their work on feedback provisioning. They extended the QVT-relations language with annotations for describing design variation points, and provided a feedback-driven backtracking capability to enable engineers to explore the design space. While this work is concerned with the performance implications of choices of per-inheritance-hierarchy OR mapping strategies, it does not attack the problem that we address, the automation of dynamic analysis through synthesis of design spaces and fair loads for comparative dynamic analysis.

The other relevant thrust of research has focused on mapping UML models enriched with OCL invariants into relational structures and constraints. Among others, Heidenreich et al. [46] developed a model-driven framework to map object models represented in UML/OCL into declarative query languages, such as SQL and XQuery. While Heidenreich et al.'s approach concentrates on mapping OCL invariants into an implementation-level language to enforce semantical data integrity at the implementation level, Trademaker automatically generates database schemas mainly based on structural constraints. Badawy and Richta [47] provided some rules guiding derivation of declarative constraints and triggers from OCL specifications. These two research work are complementary. Extending the same line, Al-Jumaily et al. [48] developed a model-driven tool transforming the OCL constraints into SQL triggers. Demuth et al. [49] also discussed a number of different approaches to implement OCL-to-SQL mapping, and developed a tool that transforms each OCL invariant into a separate SQL view definition. Different from these research efforts transforming an object model to a single counterpart in relational structures, Trademaker generates tradeoff spaces of object-relational mappings with focus on structural mapping alternatives, rather than transformation of integrity constraints.

Generating test loads. Numerous techniques have been developed for database

test generation generating testing loads [32, 50–52], including the generation of realistic loads for TPC benchmarks [53]. Among others, Khalek et al. proposed a query-aware test generation technique, called ADUSA [32] that relies on a constraint solver. Given a database schema and an SQL query as inputs, ADUSA then exhaustively generates non-isomorphic test databases. Similar to many other techniques including ours, ADUSA uses a constraint solver as a test generation engine. However, our work is different in that no prior technique generates common test loads over spaces of alternative schemas. Doing this requires enforcement of abstract design constraints as well as constraints implied by concretization mappings for each alternative. Trademaker, to our knowledge, is the first tool with this capability.

Tradeoff analysis. The other relevant line of research focuses on tradeoff analysis. Petke et al. [54] used Genetic Improvement techniques to transplant code from one version of a system to another, and then profiling the new system to enhance execution performance. Bondarev et al. [55] proposed a framework, called *DeepCompass*, that analyzes architectural alternatives in dimensions of performance and cost to find Pareto-optimal candidates. Their approach, however, requires a manual specification of architectural alternatives, and provides no support for synthesis.

Aleti et al. [56] developed *ArcheOpterix* for optimizing an embedded system's architecture. They applied an evolutionary algorithm to optimize architectures modeled in the AADL language with special focus on component deployment problems. Martens et al. [57] also developed *PerOpteryx* to automatically improve an initial software architectural model through searching for Pareto-optimal solution candidates. They applied a genetic algorithm to the Palladio Component Models of given software architectures. Like many other works we studied, these efforts do not support automatic production of design tradespaces from formal yet incomplete specifications. Rather they start from a point solution and gradually improve it. We see these two approaches for tradeoff analysis as being complementary.

Along the same lines, Trademaker [34] studies whether relational logic model

finders, such as Alloy, can be used for practical design synthesis and tradeoff analysis, and how to synthesize test suites for fair performance analysis of the resulting systems. Astronaut formalizes the notion of tradeoff analysis and contributes a theoretical framework conceptualized in dependent type theory, from which a polymorphic implementation framework for tradeoff analysis tools is mechanically derived.

Search-based software engineering. Harman [58] reviewed the application of search and optimization in eight software engineering domains, as well as open problems and challenges. Weimer et al. [20] used search techniques to help find candidate code snippets to repair buggy code. Jia and Harman [59] used automated search-based techniques to find rare but valuable test cases. McMinn [60] surveyed the application of search techniques for test data automatic generation. Our work uses search related techniques to find Pareto-optimal solutions.

3.9 Conclusion

This chapter explored how we have advanced the state of the art in searching design spaces to find system designs that improve performance related system properties. It makes several contributions to the science and engineering of software-intensive systems: a mathematical and implementation architecture for formal, automated *dynamic analysis of tradeoff spaces*; a principled approach to *load concretization* for specializing common loads to large numbers of variant implementations; experimental validation of (simple derivatives of) published ORM metrics—to our knowledge the first experimental evaluation of ORM metrics; and Trademaker-ORM, an accessible and functional tool enabling tradeoff analysis in large design spaces for the particular domain of object-relational mapping, and a testbed for ongoing research of the kind reported in this chapter. This chapter also contributes to our broader research program, which is increasingly focused on specifying, validating, realizing, and certifying

acceptable tradeoffs among non-functional properties, which remains a research challenge of the first order.

We also presented Astronaut, a novel, general-purpose software technology for practical tradespace analysis. The key contributions of this work are (1) a theoretical framework conceptualized in constructive logic to make the notion of tradeoff analysis precise, (2) a mechanically derived, polymorphic implementation framework for tradeoff analysis tools, (3) results from experiments in the domain of object-relational database mapping, corroborating the claim that widely-used industrial ORM tools appear to produce solutions with performance properties that are far from those achievable through more systematic design space modeling and analysis, and that systematic dynamic tradespace analysis can find much better designs within practical time frames, at least for modest-scale but often still useful systems.

The empirical study shows that our approach can find much better database schemas, in both time and space performance, for various ORM models of multiple applications under two common DBMSs. The Pareto-optimal schemas found by our approach completely dominate those generated by Rails and Django. Such results supported our claim that our tradeoff space analysis approach can outperform widely-used tools.

Chapter 4

System Performance Prediction with Semantic Meanings

In this chapter, we present our approach to improving the accuracy of performance prediction models of complex systems. Our approach uses the semantic meanings of configuration parameters to clean training data. We then compare the accuracy of trained models using data sets with and without our approach.

Most modern software systems are highly configurable. Configuration parameters are designed to control a software system's functional properties, but they also have an impact on non-functional properties. Engineers can adjust parameter values to adapt a system to underlying hardware and other conditions. However, it's difficult to understand how different settings affect a system's performance.

The MapReduce framework [39] is a representative system with many hundreds of configuration parameters. Since its appearance in 2008, MapReduce has become the de facto standard for implementing large-scale distributed programs to process large data sets in parallel. It hides many details of distributed computing

and only exposes simple interfaces to users. A developer can easily write a MapReduce program and submit it to a cluster to perform diverse types of computation like log analysis, Web indexing, etc. without knowing the low-level details like resource allocation, assigning parallel tasks to different machines and gathering results from each task, etc.

In the e-commerce industry, e.g., involving Walmart's online systems, MapReduce jobs serve various purposes including Search, Ads, and Personalization. Such jobs have a direct effect on customer satisfaction. Therefore, before deploying a MapReduce program onto production servers, developers usually need to run a sequence of experimental jobs to find a proper configuration. Each execution will trigger a significant amount of resource consumption due to the nature of MapReduce jobs processing large data sets. This is the most obvious difference compared with developing traditional single-machine programs, which allow for experimental execution at much lower costs.

Not only such experimental executions require tremendous resources, but most turn out to be useless to business. Jim Manzi [61] wrote in his book that only about 10 percent of such experiments were leading to business changes at Google. Moran [62] also wrote that at Netflix, 90% of such experiments are unproductive.

If developers know up front the performance of MapReduce jobs, they can optimize them without incurring the costs of actual executions. Knowing the performance of MapReduce jobs can also help the scheduler to get the best possible cluster resource utilization by placing jobs on the best available machines/containers. The B=better the job performance prediction, the better will be scheduler performance, and that means better resource utilization for a large cluster. In this project, we worked on the first step: predicting the performance of MapReduce jobs. We trained prediction models with historical execution data to predict different performance like CPU time and physical memory consumption.

Hadoop [1] is an open-source implementation of the MapReduce framework. In today's practice, people usually submit such jobs using systems like Hive. Such

a system allows engineers to write SQL-like queries which are then translated into MapReduce jobs.

In this project, we aim to predict the CPU time and physical memory consumption of MapReduce jobs submitted to the same cluster using the configurations of the cluster and the performance characteristic of Hive queries themselves. Some previous work studied the qualitative relationship of configuration parameters of MapReduce jobs and performance [18, 63, 64]. However, none to our knowledge study their quantitative relationship. We know that a configuration has an impact on final performance, but we don't know how. Previous work also did not consider job complexity as a feature that can help predict performance.

In this project, we aim to improve the prediction accuracy of performance models for big data computations. We learned how much impact individual configuration parameters can have on the performance from historical execution logs of Hadoop jobs. We also found that almost all previous work considered configurations of Hadoop only, but ignored the other sub-systems in the Hadoop ecosystem. For example, many companies are using Hive [65] to run MapReduce jobs. Users to write a MapReduce job with HiveQL—a SQL-like query language. We found that not only the core Hadoop but also other sub-systems have important impacts on performance. Instead of studying Hadoop only, we treat a MapReduce cluster as a large system combining a number of distinct sub-systems.

To improve the accuracy of trained prediction models, our approach is to use the semantic meaning of configuration parameters and job complexity measures to pre-process training data. Previously, such data were treated as pure numbers only. However, in practice, those numbers have real-world meaning. For example, the value *-1* of parameter *mapreduce.job.jvm.numtasks* means that there is no limit on how many tasks to run per JVM. This meaning is lost if we take the value as a mere number. Another example is that if we set *mapred.output.compress* to *False*, then *mapred.output.compression.codec* and *mapred.output.compression.type* will have no influence on performance because the compression functionality has been disabled. By considering actual semantic meanings, we can pre-process raw data to increase its information density to improve training results.

In this chapter, we report the results of an empirical study on the quantitative relationship between job configurations and complexity and performance in Hadoop ecosystem. We collected data from a production Hadoop cluster at *WalmartLabs*. It is part of *Walmart Global eCommerce*, which supports all information technology related business, such as shopping website `walmart.com` and `samsclub.com`. The MapReduce jobs includebut are not limited to website visiting history analysis, user shopping history analysis, and product recommendation. Their logic is spread over various sorting, regression, classification algorithm, and so on. The performance measures we studied included CPU time and physical memory consumption. We report the results of a statistical learning study about predicting the performance of MapReduce jobs running on a shared cluster.

Overall, we found that there are 49 features out of more than 900 hundred parameters that have significant impacts on both CPU time and memory consumption. The size of the input data has the largest impact on CPU time, and the number of reducer tasks has the largest impact on memory consumption. Our learned model achieves high accuracy with an R^2 score around 0.8929 in CPU time, and 0.9776 in physical memory prediction. Previous work [13] achieves only 0.6157. To check the validity of using the trained models to predict the performance of new jobs, we also run cross-validation in both cases, and the accuracy is around $0.887(+/-0.054)$ in CPU time and $0.937(+/-0.058)$ in memory consumption.

Although we just reported the prediction results for execution time and memory consumption, our approach can also predict any other resource consumption reported by the job tracker.

The rest of this chapter is organized as follows. We introduce the background of MapReduce in Section 4.1. Related work is discussed in Section 4.2. In Section 4.3, we talk about the methodology we used to carry on the study. We report the results and answer two research questions in Section 4.4. At last, we conclude in Section 4.5.

4.1 Background of MapReduce

The volume of data has been growing rapidly in almost every industry since the past decade [66]. This huge data size and the rich information involved in it change the way to store it, as well as to perform any computation over it—it is no longer possible to use a single machine due to its limited storage and computation power. Traditional High-Performance Computing (HPC) [67] cluster is not a good solution either because HPC aims to solve computationally expensive problems faster with more computing power. However, most of the emerging big data problems, such as log mining, web indexing etc. do not need much computation power. Instead, they can be easily divided into multiple smaller computation units that can be run in parallel even in off the shelf commodity machines. Thus, it becomes necessary to (i) conduct less complicated computation on smaller partitions of the whole data set and then merges the results, and (ii) send a program to the data rather than sending the data to the program, as the volume of data is much larger. To address such problem, Google introduced its MapReduce framework [39]. It assumes that a large problem can be divided into smaller ones and each small problem can be solved by one commodity machine. Thus, for a larger data set, many commodity computers are required.

Apache Hadoop [1] is an open-source implementation of the MapReduce framework. While the Hadoop File System (HDFS) [68] helps to store data in a distributed way across many machines, the programming model of MapReduce exposes very simple interfaces to implement the core programming logic by hiding the details of how data is stored and how the programs are executed in a distributed fashion. A full ecosystem is developed with many other components including Hive [65], Pig [69], Zookeeper [70], etc. to enable developers to focus on the core business logic. For example, Apache Hive is an open source infrastructure for querying and analyzing large-scale data. Hive comes with a SQL-like query language, called Hive Query Language (HiveQL), for querying data stored in a Hadoop cluster. Figure 4.1 shows an end-to-end system of a Hive+Hadoop infrastructure.

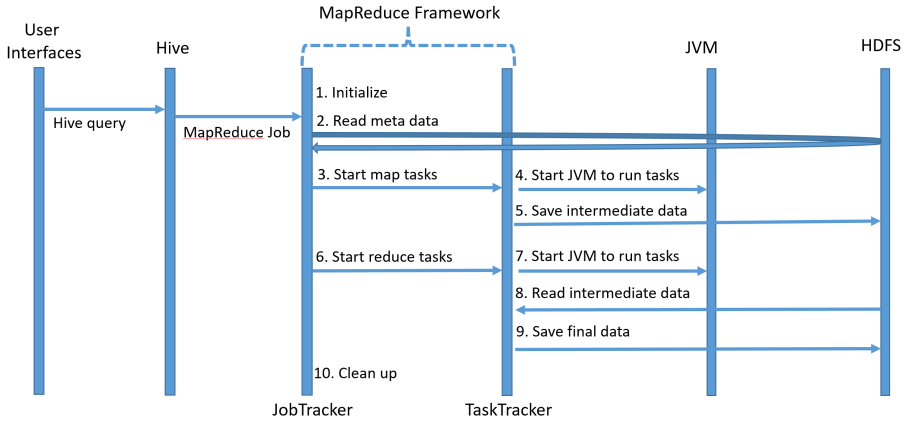

Figure 4.1: Hive-Hadoop Architecture

Figure4.1 shows the main steps of a typical Hive+Hadoop cluster setup to run a MapReduce job. The left-most layer is the user interface that allows a user to submit a Hive query. It could be a command line tool, a web UI, or APT like JDBC. Hive then compiles the query to a map-reduce job consists of a set of map and reduce tasks, and wrap them in XML files. In the context of the MapReduce framework, the typical logic phases of a job are as following: STARTING, MAP, SHUFFLE, SORT, REDUCE, and CLEANUP. The life cycle of a MapReduce job starts from the moment it's submitted to the JobTracker. In step 1 the JobTracker initializes the job, read necessary configuration files. Step 2, the JobTracker will first retrieve metadata to computes how many chunks the data will be divided into. Step 3, the JobTracker computes how many map tasks will be needed, and then sends the mapper program to the machine where the data is located. Step 4, a map task starts a JVM to execute its own logic to transform its data split to some intermediate $< key, value >$ pairs. Step 5, the map task saves the generated data in its memory (or into HDFS if the size is too large for the memory to hold). Step 6, the JobTracker starts a number of reduce tasks according to related settings. Step 7, each reduce task starts a JVM to run its logic. Step 8, a reduce task reads/shuffles some map tasks' output data and sorts them by key to $< key, list(value) >$ pairs. Step 9, the reduce tasks run its own logic to transform the $< key, list(value) >$ pairs to final $< key, value >$ pairs. At the last step 10, the JobTracker does cleanup job.

In practice, a MapReduce cluster is equipped with multiple sub-systems, with the

MapReduce framework as its core. The overall system illustrated in Figure 4.1 contains three sub-systems: Hive, MapReduce, and HDFS. Hive is used to submit MapReduce jobs with its Hive QL, which is familiar to most SQL developers. HDFS is used to store data in distributed form, and MapReduce is the framework that performs the actual distributed computing over the data. To make our discussion easier to understand, here we define some terms we will use to the left of this chapter.

Job An execution of a MapReduce program.

Component
A sub-system that works with other sub-systems to execute a MapReduce program.

Feature A single job or cluster configuration, which is an independent variable in our learning model.

Dependent Relationship
A tree-structure relationship among components and features. For example, *hive.query.string* is a child of the *Hive* component.

Background of some configuration parameters. Here we talk about the background of some important parameters found by our learning model. *mapreduce.job.reduces* is a parameter that sets how many *reducers* there will be in a job. According to Hadoop's official document, a larger value increases the framework overhead, but also improves load balancing and lowers the cost of failure. In practice, it's hard for developers to set an optimal value for this parameter. *mapreduce.input.fileinputformat.split.maxsize* sets the maximum size chunk that map input should be divided into, which indirectly decides how many mappers will be started since Hadoop starts a mapper for each chunk. *hive.exec.dynamic.partition.mode* is a parameter that controls the way to partition the data. When it's set to "strict", the user must specify at least one static partition. Otherwise, all partitions are allowed to be dynamic in "nonstrict" model. It affects the performance of a Hive job through how the data spread across a cluster. If the user knows the data well in advance, she can set the

values to be partitioned at. Dynamic partitioning could result in data skewness problems. For example, it's possible that 90% of the data belongs to one partition and the rest is spread across multiple partitions. Then one reducer will be heavily loaded and the time required to finish the whole job will depend solely on this reducer. This will significantly increase the overall execution time.

4.2 Related Work

Most software systems are configurable. The end users can tailor such a system to meet their own requirements and objectives. However, knowing how configuration affects the performance of such a system is not easy. It's possible that the configuration parameters interact with each other, and the whole configuration is generally exponential in size in relation to the number of configuration parameters. In their work on performance influence models [14], Siegmund et al. used step-wise linear regression to derive performance-influence models for a given configurable system, which can provide users the description of possible impacts of all configuration options and their interaction. They added predictive features hierarchically to the learning algorithm. Their approach only considers binary and numeric configuration options. However, each subsystem, in our case, comes with many configurable parameters, and selecting only one (or few) sub-systems while completely ignoring others in a multi-layered infrastructure does not represent the real scenario. Although their approach is generic to all configurable systems, it requires many measurements to profile a configuration space, which is not suitable for big data system due to each measurement consuming considerable resources. Meinicke et al. [71] explored the configuration complexity of highly configurable systems using variability-aware execution, which checks the same variable location in a program affected by two different configuration settings.

In other work [8,10,72], several automatic ways are presented to detect performance-related feature interactions, to improve the accuracy of predictive models. Meinicke et al. [71] conducted dynamic analysis to study the complexity of configuration spaces. They found that the configuration spaces of the studied

Java programs were not as large as expected, but that state-of-the-art strategies still cannot handle such large spaces. In our project, we used domain knowledge to select features that influence the performance of a given execution. We first studied the semantic meaning of features and then identified relationships/interactions among them. We then reduced the space based on the actual meaning and the relationships. Besides, most configuration options have been studied in other work and summarized in [18]. All those options are classified by the parameter level, MapReduce phase, and workload characteristics. This work gives us enough information to determine the features that are performance related.

In the general system performance prediction field, Zhang et al. [12] formalized each system configuration parameter as a boolean value and the overall system performance as the output of a boolean performance function that takes all configurations as inputs. They then formalized the problem of system performance prediction as learning the Fourier coefficients of a function f that transformed from the boolean performance function. However, not all configuration parameters are boolean in real-world systems. Their approach cannot predict the performance of a system with other types of configuration parameters, such as numerical, float, and string. Venkataraman et al. [73] built a performance prediction framework to analyze the performance of a shared cloud computing infrastructure. They derived the performance model by running a set of small samples of data. In our case, MCS reports details of a MapReduce job, including the static configurations and dynamic load of a job processed. In our work, we collected such information from the job tracker in Hadoop. The advantage of our approach compared to the work in [73] is that we created no overhead to the production cluster.

There is also some work in performance prediction for the MapReduce framework. Bonifacio et al. [18] conducted a systematic review of related research work in Hadoop MapReduce configuration parameters and system performance. They identified that there are 29 parameters in the Hadoop system that are related to system performance. In contrast, we collected a more broad set of parameters spread across multiple components of a MapReduce cluster, including MapReduce

and Hive, as well as JVM, IO, DFS, and so on. We found that although some parameters are clearly performance related, their values are typically set to the same across all jobs. Therefore, with no variation in settings, they carry no information to help predict performance as a function of configuration. Barbierato et al. [74] proposed to use a modeling technique to evaluate the performance of Hive based applications. This work mainly focuses on the translation of Hive queries to map and reduce tasks. Our work, on the other hand, focuses on performance prediction of MapReduce jobs as a whole, including data retrieval and computation phases. Song et al. [13] also presented a performance model to predict performance for Hadoop jobs. They profiled the features of a job by dynamically executing the job on a small sampled data. The advantage, compared to our work, is that they can catch computation complexity with dynamic execution more precisely. The disadvantage is that it increases overhead on the cluster. Besides, they worked on a small set of features: five parameters in map phase and six for the reduce phase. Because of the specific features related to execution time, their approach cannot be applied to predict other performance metrics like memory in our work.

Zhang et al. [75] presented a way to model performance of MapReduce jobs in heterogeneous cloud environments. Our work aims to predict the performance of MapReduce jobs running on a specific cluster. Khan et al. [76] proposed a Hadoop performance modeling technique which targets to predict the required amount of resource requirements if a job has a deadline requirement. Our prediction model is a more generalized approach that can be theoretically used to predict any performance that reported in job running status. In this work, we just predicted CPU time and memory consumption.

4.3 Methodology

In this section, we talk about how we collect training data and pre-process it by leveraging the semantic meanings of configuration parameters and job complexity.

4.3.1 Approach Overview

Our purpose is to build a performance prediction model that uses data size, job complexity, and cluster configurations as features to predict the performance of MapReduce jobs. The first step is to collect training data, which are Hadoop execution logs in this case. The second step is to clean the data by leveraging semantic meaning of configuration parameters, job complexity approximation, as well as standard data pre-processing techniques. Finally, we train a model to predict target performance, like CPU time and physical memory usage. The Figure 4.2 shows the overall approach we adopted to train a performance prediction model. Different with common prediction model training work, our approach leverages the semantic meaning of the features (parameters in our context) and job complexity to pre-process data.

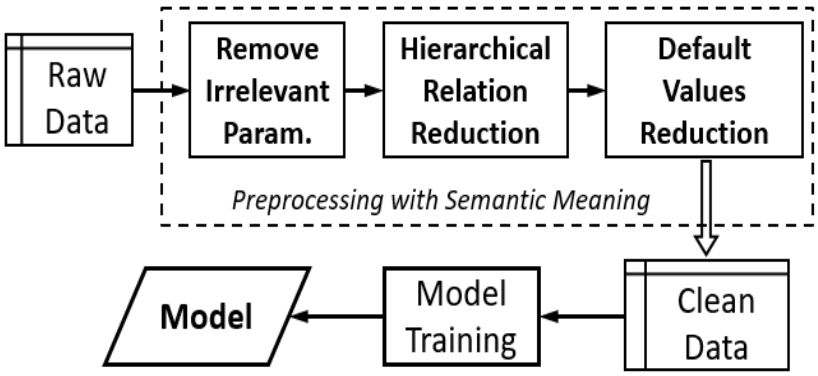

Figure 4.2: The overall approach of training a performance prediction model

4.3.2 Data Collection and Parsing

We collected job running status and job configuration web pages from the job tracker. For each job, we collect two types of raw data: (i) Job configuration details: all the configuration settings across all the layers that are used while running the job; these are our independent variables in the prediction model, as defined in 4.1. (ii) The detail of the job running status: the progress status (whether it is finished or still running) and performance of the job including (i) CPU millisecond that elapsed when executing the job (CPU Time); and (ii) the physical memory bytes consumed (Memory Consumption). These performance

Table 4.1: Example dataset including performance and job configuration

	Performance	**Configuration**			
	CPU_Time	#maps	#reduces	...	map.java.opts
job1	3621870	21	25	...	-Xmx4G
job2	1986870	84	22	...	-Xmx8G
job3	546540	18	1	...	-Xmx4G

metrics are dependent variables in our prediction models. The job tracker of a Hadoop cluster contains necessary details about a MapReduce job, from the location of data to the running status of the map and/or reduce tasks. The cluster that we collected data from has MapR Control System (MCS) [77] installed. MCS provides Hadoop administrators a centralized place to configure, monitor, and manage a Hadoop cluster. It represents a lot of information as web pages, including job configurations and running status. To collect web pages from MCS, we use a web page crawler to crawl MCS's web pages based on some URL patterns. The collected data was stored on the local disk as .html files for further processing.

We collected the data by running the crawler on a production cluster. This cluster runs different kinds of jobs every day. We collected the data in five different days randomly spread across three months—this was necessary because if all the data were in a single day, the resultant corpus might be biased by the special type of jobs that ran on that day. By collecting data from different days, we expect that the final data samples are diverse enough to cover a variety of features, and we have enough data samples for each feature.

In total, we collected 13212 data samples. We save the parsed data in an Excel file. Table 4.1 shows some example jobs. The first two columns are the CPU time and physical memory consumption extracted from the running status pages. The left columns are some example job configurations extracted from the job configuration pages.

Once we collected all the HTML files from the cluster, we parsed these files to extract the necessary information. Each job is associated with two types of HTML files containing configuration and running status respectively; for a given HTML file we first figure out which type of file it is. Then we parse it

by recognizing different HTML tags to extract configuration or status related information respectively. From the job configuration file, we extract all job settings and their values and store them in an in-memory database indexed by job ID. From files containing the running status of the jobs, we first checked whether the job is still running; we filter out all the unfinished jobs. Then for the remaining finished jobs, we extract information like the CPU time and memory consumption. We also store this extracted information in an in-memory database indexed by job ID. Finally, we merge the two information based on unique job ID. The final data is saved in Excel file format, with each row corresponding to a unique job ID and each column is a feature, either representing configuration setting or performance.

The data preparation infrastructure is primarily implemented in Python. We use Python package Beautiful Soup [78] to parse the collected HTML files.

4.3.3 Data Pre-processing with Domain Knowledge

The core of a training a prediction model is to pre-process the training data so that the trained model can achieve its potential high accuracy. The statistical analysis and machine learning research community have been studying data pre-processing techniques for a long time. Many of these techniques have become the standard approaches in pre-processing raw training data, like data integration, transformation, and feature reduction.

Instead of dealing with the training data as a set of abstract values, we argue that data gathered from real-world scenarios carries practical meaning or domain knowledge. Therefore, besides those standard data pre-processing techniques, we also leverage the semantic meaning of configuration parameters to pre-process training data. We study how parameters are used in practice and the potential relationships among them. Such knowledge helps us to identify parameters that have the potential influence on system performance so that we can remove others that clearly have no impact. The semantic meanings we leveraged are discussed in the following sections.

As shown in Figure 4.1, a typical end to end big data infrastructure consists of several components. Each of these components comprises of one to many features and each feature has many configuration options. Therefore, the configuration space of a system like Hadoop is huge. Suppose there are n parameters in a system and all of them are binary options for simplicity, then the theoretical size of the configuration space is 2^n. In reality, configuration parameters can be binary, categorical, numeric, and more. Such diversity increases configuration spaces dramatically. For a prediction model built for such a huge space from such diverse features is often hard to achieve high accuracy. It is important to identify the important components and features that actually affect end-to-end behavior, and are mutually independent with minimal interaction with each other. Thus, data pre-processing to identify these components and features become the most critical part of this work. The algorithm we used to pre-process data is shown in Algorithm 3. The semantic meanings we leveraged and specific steps to clean training data are discussed in the following sections.

Semantic Meaning#1: Irrelevant Parameters. Although a complex software system typically has many configuration parameters, many of them are not related to system performance. From Hadoop's official documents (four default configuration .xml files listed online), we can know that Hadoop version 2.7.2 has 952 parameters in total. This is a huge feature space. Luckily, many of them do not have an impact on performance. For example, "java.runtime.version" is a parameter specifying the version of Java Virtual Machine. It might influence how developers should implement a MapReduce program given the slightly different APIs in different versions of JDK. But it likely does not influence the system's performance significantly. Therefore, the first class of semantic meaning we leveraged is that some parameters are not related to system performance. We first studied all Hadoop's parameters to understand their function in the system. Then we removed those that clearly have no relationship with the performance. This step greatly reduced the number of parameters to consider.

Semantic Meaning #2: Dependent Parameters Relationship. Standard feature reduction techniques like Principal Component Analysis (PCA) treat the correlations among features as flat relationships. The situation is different

Algorithm 3: Data pre-processing with semantic meaning

Input: RawData
Output: CleandData

```
allParameters ← getAllParameters(RawData)
for p in allParameters do
    if p == hive.query.string then
        query ← getParameterValue(p)
        v ← getHiveQueryComplexity(query)
        updateParamValue(p, v)
    end if
    if hasDefaultValue(p) then
        updateParamValue(p, 0)
    end if
    if isJVMOpt(p) then
        v ← getParamValue(p)
        numV ← parseXmx(v)
        updateParamValue(p, numV)
    end if
end for
iParams ← getIndpParams(RawData)
for ip in iParams do
    dParams ← getDepParams(ip)
    pValue ← getParamValue(ip)
    if pValue == False then
        for dp in dParams do
            updateParamValue(dp, 0)
        end for
    end if
end for
CleandData ← RawData
return CleandData
```

in the scenario of software system performance prediction where configuration parameters often have the dependent structure. They interact with each other in a dependent way to carry out system functionality. Such interactions are reflected in the execution trace data as feature interaction and correlation, which will further negatively impact the prediction accuracy of the trained model [79–81].

In the system configuration domain, some parameters are usually used to turn on/off system features, and others are used to tune features. For example, Hive has two parameters namely *hive.exec.parallel* and *hive.exec.parallel.thread.number*. The first one is used to control whether to execute jobs in parallel, and the second one is used to indicate how many jobs at most can be executed in parallel. Such a relationship can help us to pre-process the training data to improve prediction accuracy. If the value of *hive.exec.parallel* is *True*, *hive.exec.parallel.thread.number* is a parameter that apparently has impact to the final performance. In contrast, if the value of *hive.exec.parallel* is *False*, *hive.exec.parallel.thread.number* will not have any impact on the final performance, because the parallel execution feature is turned off.

There are many groups of parameters which have such dependent relationships. A parameter which decides the function of some other parameters is called an "Independent Parameter". A parameter whose function is decided by an *Independent Parameter* is called "Dependent Parameter". In data pre-processing, it is important to ignore the dependent parameters like *hive.exec.parallel.thread.number* when the independent parameter disables a feature that the dependent parameter configures.

Semantic Meaning #3: The Default Value of Parameters. In the system configuration scenario, it is not uncommon that some parameters are set to the default values. When a parameter is set to its default value, it generally means that this parameter has no control over system behavior, and thus it has no impact on system performance as we discussed before. In practice, default values could be 0 in some cases, or -1, or some other values. For example, one parameter in the Hadoop system is *mapreduce.reduce.memory.mb*. Its default

value is -1. From the official document, -1 means this feature is disabled. In the data pre-processing step, we should set the value of such parameters to 0 because the system did not consider it in job execution, and thus it has no impact on the final performance. Therefore, such parameters with default values should not be considered as an effective feature in model training.

Removing low-influential parameters. In this step, we reduce the dimension of a configuration space by removing some parameters if their performance influence is lower than a threshold t. The performance influence is defined as follows:

$$pi = \sum_{k=1}^{m} avg(\sum_{i=1}^{n} \frac{abs(pd - pa_{ik})}{100})$$

, where pi is the performance influence of a parameter, pd is the performance of the default setting, and pa is the performance of alternative settings. We test the performance influence on k different benchmarking jobs n times.

To evaluate the influence power of parameters, we first create alternative values for each parameter. Table 4.2 shows some rules we used to generate alternative values. We list these rules based on parameter data types.

Table 4.2: Alternative parameter values generation rules

Data Type	**Generation Rule**
Boolean	True or False depend on the default value.
Categorical	All values specified in the official documents.
Numerical	N values around default one with Δ as [def/10]. Special case: if the default value is 0 or -1, the alternative value could be set by hand.
String	Options with random string values are unlikely. So we don't consider this case.

We first benchmark the Hadoop system with the default setting. Then, for each alternative value of a parameter, we create a new setting with other parameters fixed. Next, we benchmark the system under this new configuration and collect the performance data. At last, we compute the performance influence of a parameter using the above formula. In the end, we rank all parameters and keep top-N ones as our important parameters.

Other Special Cases. The configuration parameters in software systems have some special cases that need to be considered in data pre-processing step. We

discuss two of them here.

1. Categorical Parameters to Numerical Values. A large partition of configuration parameters are categorical values. Their values are usually restricted to a small set of predefined values. For example, *mapreduce.output.fileoutputformat.compress.codec* in the Hadoop system has 5 possible values: *GzipCodec, DefaultCodec, BZip2Codec, LzoCodec, and SnappyCodec*. We replaced such values with their category indexes. For example, if a data sample's value is *org.apache.hadoop.io.compress.GzipCodec*, then its category index is 1. There are five categories in total in this case, so the transformed values will be from 1 to 5.

2. Special String Values. Some parameters have the string type. For example, Java memory related parameters usually have values with the pattern like *-Xmx1024m*. We convert such values to a numeric value by extracting the numeric part, like the 1024 in *-Xmx1024m*.

4.3.4 Approximating Job Complexity

In today's practical big-data computing environment, Hadoop is the most basic component to store data and to perform the actual computing framework. Many other systems are built based on it to provide rich functionality. The company we worked with uses Hive, which allows users to submit MapReduce jobs using SQL-like queries. The logic complexity of a Hive query determines things like how many map and/or reduce tasks to start, which could impact the final performance a lot. However, it is hard to tell how complex a query is in terms of how significantly it impacts a job's performance. In this project, we use a workaround solution to approximate the complexity of a Hive query.

Given a Hive query, Hive first parses it to an abstract syntax tree (AST), and then convert into a dependency graph of MapReduce tasks. The number of vertices and edges of the graph depend on the logic complexity of the give Hive query[1]. For example, a simple query could be like "SELECT col1, col2 from table1", which retrieves two columns from a table. The dependency graph for

[1] https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Explain

this query is very simple. There is even no "reduce" task needed, only map tasks. Actually, the shape of the dependency graph is determined by information in a query like the number of tables, columns, and other data processing operations like *groupby*, *union*, *orderby*, and so on. Thus, we leveraged such information to approximate the complexity of Hive queries.

In order to easily extract information from a query, we first built an AST of a given Hive query using the Hive's built-in parser [82]. Then we traversed the AST to extract the number of some entities and operations in a query. The information we extracted are listed in Table 4.3.

Table 4.3: Information extracted from Hive query

table	column	select	clusterby	sub_query
join	intersect	load	leftouterjoin	leftstemjoin
where	crossjoin	groupby	rightouterjoin	fullouterjoin
sortby	unionall	orderby	uniondistinct	distributeby

The formula we used to compute the final complexity is:

$$complexity = \#table * (sum(\#all\_other\_extracted\_items))$$

Algorithm 4: Function *getHiveQueryComplexity()* to approximate Hive query complexity

Input: query: A Valid Hive Query
Output: complexity

$ast \leftarrow buildAST(query)$
$\#table \leftarrow getNumTable(ast)$
$\#select \leftarrow getNumSelect(ast)$
$\#groupBy \leftarrow getNumGroupBy(ast)$
$\#insert \leftarrow getNumInsert(ast)$
...
$\#orderBy \leftarrow getNumOrderBy(ast)$
$\#crossJoin \leftarrow getNumCrossJoin(ast)$
$\#union \leftarrow getNumUnionAll(ast)$
$complexity \leftarrow \#table * (\#select + \#insert + ... + \#orderBy + \#crossJoin)$
return *complexity*

4.3.5 Standard data pre-processing.

In the previous steps, we leveraged the semantic meaning of parameters to pre-process the training data, like removing parameters that are not related to performance. Besides, we also use some standard data pre-processing techniques, like adding missing values and feature scaling. These steps will further clean the data to speed up the training procedure, to improve the prediction accuracy, and so on.

Missing Value Imputation. Data is often dirty in practice. It is often that we have corrupt or missing values. Missing value imputation is a technique, which identifies, marks, and handles missing values to get good performance in learning models. The common steps to apply this technique are: 1) mark missing values; 2) remove data with missing values; and 3) impute missing values. In this project, we use Python's scientific computation and machine learning ecosystem to parse, clean, and train models. Therefore, all missing values are represented as NaN value in numpy [83] library. We scan the whole dataset to find NaN values to find missing values. We then summarized the percentage of missing values in each feature. There are about 5% of features having more than 20% missing values. We first remove these features. The features left will be those with less than 20% missing values. We then replace the missing data with appropriate values for different types of features. For numerical features, the three most common replacing values are $Mean$, $Mode$, or $Median$. In this work, we impute missing values with the $Mean$ value of each feature. For boolean features, we replaced missing data with $False$. For category features, we replaced the missing data with the $Mode$ of all values for each feature. Other than removing feature by missing values, we also remove jobs using a similar approach. We remove a job from the training data if it has more than 40% missing values. In this step, there are 44% features are removed.

Feature Scaling. Feature scaling is a technique to normalize data into a small range. The purpose is to make the training algorithms converge and run fast. In this project, we scaled all data into the range [-1, 1]. The formula we used

is:

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

.

4.3.6 Model Selection and Training

In this section, we will talk about how we choose a machine learning model and how to train the prediction models for CPU cycles and performance prediction.

Model Selection. To select an appropriate model, we first need to understand the training data. In our case, they are the configuration parameters of jobs that successfully executed on a cluster, as well as their performance related characteristics. In MapReduce performance prediction scenarios, the training data has two characteristics. The first is that the data collected in two or three months might not cover all jobs. It's very possible that some new jobs will be submitted to the cluster in the future. It's good to have the ability to predict performance for those new jobs. The second characteristic is that the data are possible imbalanced because some jobs were executed multiple times on different input data, but others were executed only a few times. Because of these two characteristics, a model that is not sensitive to the imbalance in the training data is preferred.

Random Forest is a machine learning model that uses bootstrap as its internal sampling technique, which is not sensitive to the imbalance in the training data. Moreover, the configuration space here has the nature of tree structure. For example, if we treat the root of the tree as the final performance, its value is actually determined by many children like Hive and Hadoop. Figure 4.3 shows the child tree with "Hive" as its root. The performance of this tree is determined by 5 other child trees like "compactor" and "exec". Similar, the performance of the "Hadoop" tree could be impacted by JVM options and/or the number of splits of the input data.

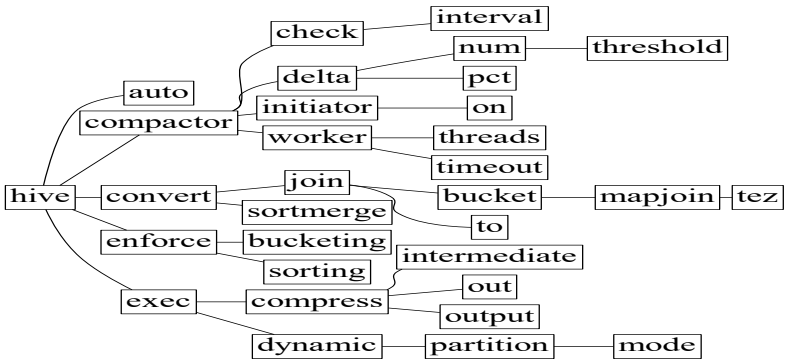

Figure 4.3: Example of feature dependency

Therefore, we choose Random Forest as the machine learning model in our MapReduce performance prediction task. We also tried a few other models: Linear regression, polynomial regression, and support vector machine. In section 4.4, we mainly report the results of Random Forest model, and introduce results from other ones we have tried.

Model Training. We trained two regression models for CPU time and memory consumption respectively. We used the random forest implementation in Scikit-learn [84] to train the models. When training the model, we used 67% of the whole data set as training data, and 33% as the test data. When doing cross-validation, we used 5-fold splitting strategy.

4.3.7 Semi-auto dependent relation discovery

In previous sections, we discussed how to leverage the semantic meanings of configuration parameters to pre-process the training data. Here we talk about how to semi-automatically discover such semantic meanings.

In order to leverage parameters' semantic meanings to pre-process the training data, we first need to find out those meanings. Here we talk about how to automatically discover the dependent relationships among parameters. As we mentioned before, boolean parameters are usually used to disable/enable system features, and the other types of parameters are used to tune features. Given a boolean parameter with a pattern like "AAA.BBB", we search for all other

parameters with the pattern like “AAA.BBB.CCC” where “CCC” could be a string with multiple dots. Here “AAA.BBB” is an independent parameter and “AAA.BBB.CCC” is the dependent parameter. All these discovered dependent relationships are stored in a .json files so one can validate them. Users can also define extra items based on their understanding of a system. An example dependent relationship item in a .json file is like Listing 4.1. The value 1 of “type” property means the type of this item is a dependent relationship. The “IndpParam” property is the independent parameter. The “DepParams” are dependent parameters, and its value is an array.

```
{
    "type": 1,
    "IndpParam": "fs.s3a.multipart.purge",
    "DepParams": ["fs.s3a.multipart.purge.age", "fs.s3a.multipart.size",
        "fs.s3a.multipart.threshold"]
}
```

Listing 4.1: An example of dependent features definition

The “Default Value” semantic meaning is hard to be discovered automatically, due to the default values for different parameters could vary a lot. For example, some parameters might have -1 as their default value, and other ones might have 0. Therefore, users need to define the default value for each parameter. Similarly, one should store the defined “Default Value” meaning in a .json file. The Listing 4.2 is an example item, where the value 2 of “type” meanings this is a “Default Value” semantic item, and “IndpParam” is the parameter name, and “value” is the default value for the parameter.

```
{
    "type": 2,
    "IndpParam": "mapreduce.job.reduces",
    "value": 1
}
```

Listing 4.2: An example of default value features

We have studied all parameters of Hadoop and Hive and created such .json files for these two systems. They are published through our code repository. These defined semantic meanings can be used as a reference. One can add new items

or update existing ones manually or through our tool DooPred. The changed relationships will be stored in corresponding files for later usage.

DooPred accepts two kinds of input training data. The first kind is a set of parsed job execution history, which should be stored in an excel file with each column as a configuration parameter and each row as a job execution record. The second kind is a folder that contains the raw Hadoop log. In default, Hadoop stores job configurations in .xml files, and stores the execution reports which contains performance information in .json files. To train a prediction model with DooPred, one should first collect the Hadoop log files. We suggest users collect as much data as possible, and as diverse as possible, due to a large number of configuration parameters (features) requires more samples in general.

Through DooPred, users can update various settings for data cleaning, model training, and verification. For example, one can set the percentage thresholds of missing value in configuration parameters and data samples. Such settings will affect the quality of the trained model through the number of imputed values. One can turn on or off cross-validation, and set the number of folds in K-fold cross-validation. All these settings can be given through a configuration file or be set from the settings dialog of DooPred.

The model training is simple by clicking the "Train Models" button. DooPred will train the models given by its configuration. In default, DooPred will only train random forests models. The accuracy of the trained model will be tested with cross-validation in default, and the test results will be shown in the result window. If one thinks the accuracy is not as good as expected, she can adjust the training settings and try again. The trained model can be saved onto disk for later usage.

4.4 Results

In this section, we report the prediction results in CPU time and physical memory consumption and answer two research questions.

RQ1. How efficient our model can predict CPU time of MapReduce jobs?

In this research question, we studied how our prediction model works when predicting CPU time. Specifically, we report important features recognized by and accuracy of the random forest model in Table 4.4.

Sub-RQ1: What are the important features in predicting CPU time? After data pre-processing we discussed in Section 4.3.3, there are 30 features that have more or less impact to the CPU time. Here we report top 5 and their importance reported by the random forest model in the upper half of Table4.4. We can see that the two most important features are *mapreduce.job.maps* and *mapreduce.job.reduces*, which are the numbers map and reduce tasks. The third feature is *mapreduce.input.fileinputformat.inputdir*, which is an approximation of input data. Hive splits data by meaningful indices like date and each partition ends up in a new directory. Hence the number of directories or partition counts is a good approximation of the input data. The fourth feature is *hive.query.string.comp*, which is the complexity of a Hive query.

Table 4.4: CPU time prediction: top 5 parameters and model quality

Parameter	**Importance**
mapreduce.job.maps	0.3888
mapreduce.job.reduces	0.3631
mapreduce.input.fileinputformat.inputdir	0.1203
hive.query.string.comp	0.0869
mapreduce.input.fileinputformat.split.maxsize	0.0127
Metrics	**Value**
R^2 Score	0.8929
Cross Validation	0.887 (+/-0.054)

We compared the important parameters found by our approach with the those discussed by other researchers previously, which are summarized in a review paper [18] by Bonifacio et al. Since the authors didn't explicitly mention which particular parameters are strictly related to the *time* performance, we assume that all CPU related parameters contribute to *time* performance. We found that

our approach can identify more important parameters that cannot be captured by previous work. To be specific, only one parameter was identified in that paper, which is *mapreduce.job.reduces* with 0.3631 importance value computed by our model. Previous work failed to find other parameters that contribute to the final time performance. Besides, our model identified the quantitative relation of each parameter with the *time* performance, while most of the previous work only studied the qualitative relation.

Our approach has two advantages compared to previous work. The first is that we consider the big data infrastructure in a real practice scenario. Most previous work only considers the parameters in the Hadoop system, per se. The second is that we quantitatively studied the importance of each parameter to time performance, which can provide deep understanding to cluster administrators, while most previous work only studied qualitative relationd.

Sub-RQ2: Performance prediction by random forest In this project, we use cross-validation result and R^2 score to measure the quality of the trained model. As the lower half of Table 4.4 shows, the R^2 score is 0.8929 and CV result is $0.887(+/-0.054)$.

RQ2. How efficiently can our model predict physical memory consumption?

In this research question, we report the prediction results for physical memory consumption, as well as the results of evaluating the prediction model itself.

Sub-RQ1: What are the important features in predicting memory consumption? The top 5 important features are listed in the upper half of Table 4.5. In this case, the most important feature is still the number of map tasks, and its importance is 0.7279. The second important feature is also *mapreduce.job.reducers*. Its importance is 0.1731. The third important feature is the number of input folder with importance 0.0597, which is an approximation of input data as we discussed. The job complexity if the fourth important feature.

Table 4.5: Memory prediction: top 5 parameters and model quality

Parameter	**Importance**
mapreduce.job.maps	0.7279
mapreduce.job.reducers	0.1731
mapreduce.input.fileinputformat.inputdir	0.0597
hive.query.string.comp	0.0316
fs.hdfs.impl	0.0021
Metrics	**Value**
R^2 Score	0.9776
Cross Validation	0.937 (+/-0.058)

Just as we evaluate the CPU time prediction model, we used cross-validation and R^score to evaluate the prediction model. The evaluation results are listed in the lower half of Table 4.5. The results are better than those for CPU time prediction.

Theoretically, different kinds of performances are impacted by different sets of important features. This is verified by the data in Table 4.4 and 4.5. Although the important parameters are highly similar, their importance values are actually different. For example, the importance value of *mapreduce.job.maps* increases from 0.3888 in CPU time case to 0.7279 in memory case. However, the importance of *mapreduce.job.reduces* drops from 0.3631 to 0.1731. The other parameters also show a significant difference.

RQ3. What are the impacts of job complexity and features' dependent structure?

In this research question, we aim to study how job complexity and the dependent structure of features affect the final performance of a trained model.

Table 4.6 shows the quality of trained models with and without job complexity and features' dependent structure. The first two rows are R^2 and cross-validation scores of the trained model without considering complexity and dependent structure. The next two rows are those of the model with dependent structure. And the next two rows are those two scores with job complexity. The last

Table 4.6: Model quality with and without job complexity and dependent structure

Feature	Metrics	CPU Time	Memory
None	R^2	0.7460	0.9366
	CV	0.707 (+/-0.095)	0.919 (+/-0.014)
With H.	R^2	0.7597	0.9338
	CV	0.708 (+/-0.059)	0.929 (+/-0.030)
With C.	R^2	0.8779	0.9750
	CV	0.871 (+/-0.073)	0.936 (+/-0.062)
Both	R^2	0.8929	0.9776
	CV	0.887 (+/-0.054)	0.937 (+/-0.058)

two rows list the quality of models with both job complexity as a feature and pre-process data with dependent structure. As the results show, we can obtain the best model quality when we incorporate both.

4.5 Conclusion

Performance testing for big-data jobs consumes much more resources than traditional single-machine applications. Developers usually need to run a sequence of testing jobs before finding a satisfying one. Generally, a Hadoop cluster runs production jobs in 10% of the time, whereas 90% resources are consumed by jobs which are "experimental". Jim Manzi [61] said that at Google, only about 10% of controlled experiments were leading to business changes. Mike Moran [62] wrote that Netflix considers 90% of what they try to be wrong. A predictive tool above exercise can drastically improve developer productivity and can bring down operation cost.

This chapter presents an approach to improve the performance of MapReduce jobs using semantic meanings of configuration parameters, such as dependent structure. These MapReduce jobs are started as Hive queries which is a common setting in today's industry practice. Our approach improved the model quality for both CPU time and memory consumption prediction. We also found that parameters have different impacts on CPU time and memory consumption.

These results showed that be leveraging parameters' semantic meanings, we can remove irrelevant information from the training data to improve the quality of performance prediction models.

Chapter 5

Improving System Performance via Configuration Space Exploration

An important lesson we learned while doing the work in the last chapter is that learning an accurate performance prediction model for large and complex systems is extremely hard. The question is: can we find high-performing configurations without learning a model? In this chapter, we demonstrate an approach using meta-heuristic search technique to explore configuration spaces of large systems to improve their performance. We construct our approach based on an evolutionary MCMC algorithm and compare it with three baseline approaches. We also present the advantages of our approach over such baselines, mainly focuses on how much performance improvements we can gain over them.

Many software systems are highly configurable and can be tailored to meet different needs in different environments. In practice, people often find it hard to understand how to take advantages of rich configurability, and just use

pre-packaged or *default* configurations [2]. As a result, they leave significant performance potential unrealized. Configuring a system to achieve better performance is important [85], in particular, for big data systems because "even a small percentage of performance improvement immediately translates to huge cost savings because of the large scale" [86].

However, automatically finding the optimal configuration for big data systems is often challenging as the configuration spaces of big data systems are vast: (i) One system usually contains many configurable sub-systems, which increases the complexity of the whole configuration space [87]. For example, Hadoop has around 900 parameters across 4 sub-systems. (ii) The configuration parameters can have fields with diverse types, as well as optional and dependent substructures. For example, setting one boolean parameter to *true* can enable an entire subsystem, requiring many more configuration values. Further, some parameters, especially the numeric ones, can have thousands of (or more) alternative values. (iii) Configuration parameters can be parameterized by external constraints: e.g., one cannot set a Hadoop *number-of-CPUs* parameter to a number larger than the available number of CPUs. (iv) Finally, analysts often do not know key properties of configuration-to-performance functions: e.g., how parameters interact, whether properties analogous to convexity, linearity, and differentiability, etc. For such complex and discrete problems, standard mathematical optimization methods are known not to apply well [88]. Therefore, selecting the optimal parameters from such complex configuration space is often a "black-art" [89].

Finding high-performing configurations for traditional software systems is known to be hard and has given rise to a significant body of work [8–10, 12, 14, 73]. Much work focuses on learning generalized models to predict a given configuration's performance. To learn such a model, one must profile a system under many configurations to sample the objective function under widely varying conditions. However, the cost of sampling real-world systems are usually high [90] and they often do not generalize well for complex configurable systems [11]. To address these issues, Nair *et al.* [11] proposed a rank-based model. Instead of predicting the performance of a configuration, they answer whether one configuration is better than others. They showed that such a rank-based technique requires fewer

samples than residual-based models.

However, for a much higher-dimensional configuration space, such as ours, it is hard to learn even a rank preserving model with high accuracy (see in Section 5.5). One can think of using Neural Networks (NN) in such scenario. However, NNs are known to require a large amount of data for training; the cost of collecting such training data for big-data systems can be prohibitively expensive [91]. Thus, instead of finding an optimal configuration, we reduce the problem as:

- Find a *good enough* configuration within a limited computation budget to achieve better performance.
- Find the configuration with *as little cost as possible*.

The intuition that motivated this work is that learning generalized models is perhaps unnecessarily expensive to find a *good enough* configuration. Our approach is to use meta-heuristic search methods, such as Markov Chain Monte Carlo (MCMC), genetic algorithms (GA), derivative-free methods, etc., to converge more directly on good configurations through guided search around seed configurations, without the need to learn generalized models over the entire configuration space. In this paper, we present an approach using Evolutionary Markov Chain Monte Carlo (EMCMC) methods, in particular.

To minimize the search cost, we formulated and experimentally validated two hypothesis. First, we hypothesize that for approximating the *configuration-to-performance* objective function, a 100X smaller instance of a given job can be an effective proxy. We call it *scale-up* hypothesis. Second, we hypothesize that better configurations for given jobs would yield improvements for *similar* jobs, avoiding the need for repeated optimization runs. We call it *scale-out* hypothesis.

To evaluate, we conducted a set of experiments using the Hadoop and Spark big data frameworks as experimental testbeds, and five canonical big data problems as benchmarks for each system. The experimental results show that our approach can find configurations to improve CPU time performance over the default configuration from 14.2% to 25.2%. The performance gain achieved by

the best configuration found by EMCMC based search strategy can outperform the one found by random search by 17% to 77%. We also compare ConEx *w.r.t.* state-of-the-art Nair *et al.*'s learning-based approach [11], and ConEx can gain 5.3% to 1700% (17 times) more performance improvement.

The main contributions of this part of our work are as follows:

- We present a novel approach to searching high-dimensional configuration space for big-data system to achieve better performance. The search performance outperforms the base-line system with significant margin.
- Our scale-up hypothesis testing results show that configurations found with small job instances also tend work for much larger jobs, saving significant experimental cost.
- Our scale-out hypothesis results show that configurations found with representative jobs can also improve the performance of similar jobs, thus saving significant experimental cost.

5.1 Background

This section presents the relevant background. First, we define some key terms that we use in this work.

Definition 5.1.1. *Configuration Parameter (c_i).* A configuration parameter is a variable, the value of which is set by the installer and/or user to specify how to configure a particular property of a system.

Definition 5.1.2. *Configuration (c).* A *configuration tuple*, or simply a *configuration*, $c = [c_0, \ldots, c_N]$, is an N tuple, where each element c_i is a configuration parameter and N is the total number of parameters, i.e., the dimensionality of the space.

Definition 5.1.3. *Configuration Space (ζ).* A configuration space, $\zeta = \{c|valid(c)\}$, is the set of all valid configuration tuples for a given system. The definition of *valid* varies from system to system. If there are no constraints on what it means

to be valid, and if N is the number of parameters and M is the average number of values of each parameter, then the size of ζ will be roughly M^N.

Parameter	**Tuple**$_1$	**Tuple**$_2$
dfs.blocksize	3	2
mapreduce.job.ubertask.enable	FALSE	True
mapreduce.map.java.opts	-Xmx1024m	-Xmx2048m
...	...	...
mapreduce.reduce.shuffle.merge.percent	0.66	0.75

The above table shows two sample configurations for Hadoop. We list only four parameters due to space limitations. In practice, configurations have many parameters, of varying types: boolean, integer, categorical, string, etc. For parameters with large domains (e.g., integer- or string-valued parameters), the dimension of the configuration space can be vast even when there are only a few parameters. Table 5.1 summarizes the configuration space of Hadoop and Spark.

Table 5.1: Configuration Space Characteristics

System	**Total Parameters**	**Studied Parameters**						**Total Configurations**
		Total	**Bool**	**Int**	**Float**	**Categorical**	**String**	
Hadoop v2.7.4	901	44	4	26	6	3	5	$3 * 10^{28}$
Spark v2.2.0	212	27	7	14	4	2	0	$4 * 10^{16}$

Note that, although we have studied a small subset of the total available parameters, since some parameter domain is large, the studied configuration space is still huge.

5.1.1 Heuristic Optimization

A promising approach to find optimal configuration can be based on meta-heuristic search and sampling, as shown by Oh *et al.* [92] for software product lines. Since our goal is to find a *good enough* configuration within a given computation budget, the choice of heuristic optimization strategy becomes important for faster convergence. There are many optimization techniques one could try. Gradient descent, for example, is useful if error functions are differentiable, but that is not the case here. Coordinate descent and other derivative-free techniques, e.g., stochastic cyclic coordinate descent [93], are often useful, but can get stuck in local optima [94], as we saw when we tried

these methods in our domain. Having explored a range of such methods, this work has adopted an approach based on MCMC methods, as explained below.

5.1.2 Markov Chain Monte Carlo (MCMC)

Monte Carlo is a sampling method to draw samples from a probability distribution $P(.)$ defined on a high dimensional space [95]. Markov Chain assumes that a configuration c_t, given all its previous instances $\{c_0,c_1,...,c_{t-1}\}$, only depends on its immediate neighbour c_{t-1}, *i.e.* $P(c_t|c_0,c_1,...,c_{t-1})=P(c_t|c_{t-1})$. MCMC combines the two methods; since the target distribution is unknown, MCMC operates on a proposal distribution. Given a configuration c_i, MCMC draws a neighbor c^* from the proposal distribution and evaluates its fitness value (i.e., better performance). Then, it decides to accept/reject this neighbor based on the acceptance probability of the fitness value. The acceptance probability can be computed using widely used Metropolis algorithm [96]:

$$A(c_{new}|c_{curr}) = min(1, \frac{P(c_{new})}{P(c_{curr})}) \tag{5.1}$$

If a neighbor is accepted, MCMC transits to the next state (c_{new}) from current state (c_{curr}). Thus, MCMC repeatedly samples a candidate next state, c_{new}, from a proposal distribution, accepts it with an acceptance probability, and repeats until a specified computing budget is reached [97]. Note that, due to the probabilistic nature of acceptance, it can accept some samples with worse fitness than c_{curr}, introducing diversity that mitigates problems due to noise and local minima.

As the number of accepted samples increases, the MCMC sampling distribution approaches the target configuration distribution. A good MCMC algorithm is designed to spend most of the time in the high-density region of the target distribution. Thus, MCMC is often used to sample approximate global optima. A detailed description of MCMC can be found in [98].

5.1.3 Evolutionary MCMC (EMCMC)

MCMC algorithms use a single Markov chain, often initially discarding most candidate states, thus converging slowly [95]. The *Evolutionary-MCMC* algorithm addresses this problem. Like evolutionary (e.g., genetic) algorithms (GA) [99–101], EMCMC starts with a population of $N > 1$ states, selects a subset of high fitness states, and then obtains a population of states for the next round by applying mutation and cross-over operations to the selected states. A key difference between ordinary GAs and an EMCMC approach is that in GAs, only strictly better states are accepted, whereas EMCMCs can accept slightly worse states, as explained previously.

Cross-over. Cross-over operations work by randomly selecting parent configurations, c_i and c_j. Each configuration is divided in 2-parts (1-point cross-over). Then the first part of c_i is mixed with the second part of c_j and vice versa, generating two offspring configurations.

Mutation. Given a configuration generated by the cross-over operation, c_k, the Mutation operation updates the value of some randomly selected parameters to generate a new configuration, c_k^*.

5.2 Technical Approach

Our basic approach is to use EMCMC algorithms to sample Hadoop and Spark configuration spaces in search of high-performing configurations. To reduce the cost of, or even need for, costly EMCMC runs, we employ two additional tactics.

1. **Scale-up**: We run Hadoop and Spark using inputs that are 100X smaller than those that we want to run in production for sampling performance as a function of configuration.

2. **Scale-out**: We use a measure of the similarity of big data jobs to enable what amounts to a kind of transfer learning, wherein good configurations for one job are used for another without change.

We have implemented our approach in a tool called ConEx. The rest of this section describes our approach in detail.

Overview

An overview of ConEx is shown in Figure 5.1. ConEx takes a big data job as input and outputs a "good-enough" configuration by sampling the configuration space using EMCMC strategy (see Section 5.1). ConEx works in three phases. First, in Phase-I, it filters out some configuration parameters that are not relevant to the objective function (job performance in our case). Next, in Phase-II, ConEx uses EMCMC sampling strategy to find a "good-enough" configuration. The sampling process starts with the default system configuration as the seed value. While sampling, ConEx discards any invalid configurations that may be generated during sampling using a checker developed by Tang *et al.* [102] (Phase-III). If a configuration is valid, ConEx runs the input job with this new configuration. ConEx records the performance (CPU and wall-clock time) of this execution. It then compares it with that of the current best configuration, updating the latter if necessary, as per our acceptance criterion (see Equation (5.1)). The accepted configurations are subjected to cross-over and mutation to produce the configurations for the next round of sampling. Once ConEx exceeds the sampling budget, it outputs the best configuration found so far. We now describe each of these steps in greater detail.

Phase-I: Pre-processing the configuration space

The total number of configuration parameters for Hadoop and Spark are 901 and 212 respectively (see Table 5.1). However, the majority are unrelated to performance. Moreover, all values of the related parameters are not equally relevant when searching for good results. Thus, in this phase, we reduce the

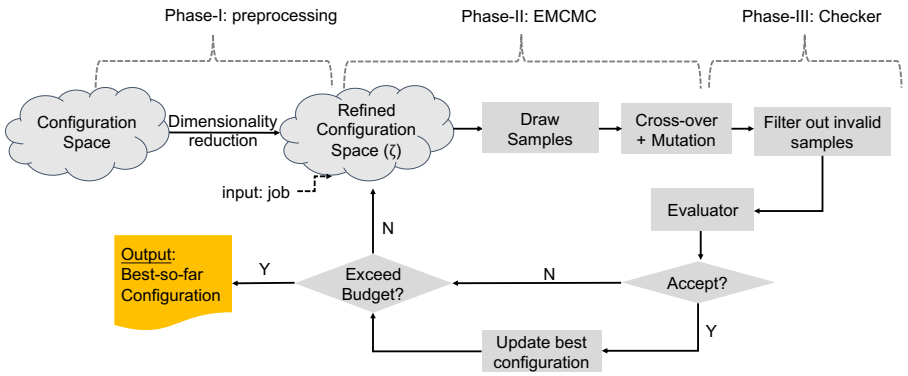

Figure 5.1: ConEx Workflow

dimensionality of the search space in two ways: (i) we consider only the parameters relevant to performance using domain knowledge, and (ii) for each selected parameters, we select only a few values for sampling.

The first part is manual; we referred to technical manuals and other work cited in this review paper [18]. For example, we removed Hadoop parameters related to version (*e.g.*, java.runtime.version), file paths (*e.g.*, hadoop.bin.path), authentication (*e.g.*, hadoop.http.authentication.kerberos.keytab), server-address, and ID/names (*e.g.*, mapreduce.output.basename). These parameters have negligible impacts on the CPU time. For Spark, we selected parameters related to the runtime environment, shuffle behavior, compression and serialization, memory management, execution behavior, and scheduling. In this way, we select 44 and 27 parameters to study further. Table 5.1 summarizes these parameters.

Note that, although this step significantly reduces the number of configurations, since some of the parameter domains are large (*e.g.* integer, float, string), the resulting configuration space is still huge: $3 * 10^{28}$ and $4 * 10^{16}$ for Hadoop and Spark respectively. Thus, even the reduced configuration space is several orders of magnitude larger than those studied in previous work. For example, most systems studied by Nair et al. [11] only have thousands of configurations, with only a few systems like SQLite have millions of configurations.

We further discretize the configuration space by defining sampling values for each parameter, varying by parameter type. Boolean parameters are sampled for only true and false values. We sample numerical parameters within a certain

distance from their default. For string-valued parameters (*e.g.*, as Java virtual machine settings) we provide lists of alternative values.

Phase-II: Finding a near-optimal configuration

This phase is the core part of our approach driven by EMCMC strategy and implemented by Algorithms 5 to 7. Algorithm 5 is the main driving function; Line 1 lists the inputs and outputs. The algorithm takes the refined configuration space (ζ) and a given job as inputs. ConEx samples configurations from ζ and evaluates performance *w.r.t.* the job. The routine also requires a seed configuration ($conf_{seed}$), and a termination criterion of the maximum number of generations (max_gen). We choose $max\_gen = 30$ in our experiment. The final outputs are the best found configuration ($conf_{best}$) and its performance ($perf_{best}$).

Algorithm 5: Explore Configuration Space with EMCMC

```
1  Function EMCMC()
   Input  : Refined Configuration Space ζ,
            job,
            seed configuration conf_seed,
            threshold max_gen, min_improvement
   Output : Best configuration conf_best,
            Corresponding performance perf_best
2  perf_seed ← run job with conf_seed
3  conf_best ← conf_seed
4  perf_best ← perf_seed
5  generation ← 1
6  Δperf ← 0
7  conf_parents ← sample n random configurations from ζ
8  do
9      confs_accepted ← EmptyList
10     foreach parent conf_p ∈ conf_parents do
11         perf_p ← run job with conf_p configuration
12         accepted ← Accept(perf_best, perf_p) # based on Eq. 5.1
13         if accepted then
14             confs_accepted.add(conf_p)
15             if perf_p > perf_best then
16                 conf_best ← conf_p
17                 perf_best ← perf_p
18     Δperf ← (perf_best − perf_seed)/perf_seed
19     conf_parents ← evolve(conf_best, confs_accepted)
20     generation ← generation + 1
21 while generation < max_gen;
22 return conf_best, perf_best
```

Lines 2 to 6 initialize some parameters including setting the best configuration and performance to the respective seed values. Line 7 gets the first generation of configurations by randomly sampling n items from ζ. We choose $n = 4D$ where D is the number of parameters, but it could be set to any reasonable value. Lines 10 to 17 are the main procedure for evaluating and evolving *w.r.t.* each configuration. Given a configuration $conf_p$, Line 11 records the job's performance ($perf_p$) and Line 12 decides whether to accept it based on Equation (5.1). This part is realized in Algorithm 7. If accepted, Line 14 stores the accepted configuration to a list $confs_{accepted}$, which is later used in generating next-generation configurations (Line 19). If the accepted one is better than the previously found best candidate, Lines 15 to 17 update them accordingly.

Once all the first generation configurations are processed, Line 18 computes the performance improvement achieved by this generation *w.r.t.* the seed performance and will be used as a part of termination criteria. Next in Line 19, the algorithm prepares to enter the next generation by generating offspring configurations using cross-over and mutation operations (see Algorithm 6). Line 20 updates the generation number. This process repeats until the termination criterion is satisfied (Line 21). Finally, the last line returns the best found configuration and the corresponding performance.

Algorithm 6: The evolutionary sub-routine of EMCMC

1 Function Evolve
  Input : $conf_{best}, confs_{accepted}$
  Output : $conf_{children}$
2 $conf_{children} \leftarrow EmptyList$
3 $P_{crossover} \leftarrow$ randomly select 50% parameters from $conf_{best}$
4 $P_{mutate} \leftarrow$ randomly select 6% parameters of $conf_{best}$
5 **foreach** $conf_p \in confs_{accepted}$ **do**
6 $conf_{new} \leftarrow crossover(conf_{best}, conf_p, P_{crossover})$
7 $conf_{new} \leftarrow mutate(conf_{new}, P_{mutate})$
8 $conf_{children}.add(conf_{new})$
9 **return** $conf_{children}$

Algorithm 6 is the evolution sub-routine of the EMCMC algorithm. It is adapted from a standard Genetic Algorithm [99]. For preparing configurations of the next generation, it takes the best configuration found so far and a list of parent

configurations as inputs. There are two main steps in the evolution process: cross-over and mutation. From the best configuration, Line 3 selects half of all parameters as cross-over parameters ($P_{crossover}$), and Line 4 identifies 6% of all parameters as mutation parameters (P_{mutate}). Next, for each parent configuration, Line 6 exchanges the values of the cross-over parameters with $P_{crossover}$. It then randomly mutates the values of the mutation parameters at Line 7. The resulting offspring is added into the children set at Line 8. A set of new offspring configurations is returned at Line 9.

Algorithm 7: Acceptance sub-routine of EMCMC

```
1 Function Accept
  Input  : perf_best, perf_p
  Output : bool: accept/reject conf_p
2 Δperf ← (perf_best − perf_p) / perf_best
3 accept_prob ← exp(50 * Δperf)
4 if Rand(0, 1) < accept_prob then
5     return True
6 return False
```

Algorithm 7 computes acceptance probabilities as described in Equation (5.1). It takes two performances as input: a current candidate under test and the best one found so far and returns whether to accept/reject the current candidate. First, Line 2 computes the performance improvement ($\Delta perf$) between the two. If the current configuration is worse than the best value, $\Delta perf$ will be negative. Next, Line 3 computes the acceptance probability of the configuration using an exponential function, which returns a positive value even if $\Delta perf$ is negative. Finally, the acceptance probability is compared with a random number sampled between 0 and 1 exclusively. Thus, even if a configuration is slightly worse than the best one, it still has some chance of being accepted.

Phase-III: Configuration Validity Checking

Configuration spaces are often not complete cross-product spaces. Rather, they are subsets defined by constraints on individual parameter values and sets of parameter values. A problem that we encountered is that constraint on configurations—at least for Hadoop and Spark, and we expect for many

systems—are not well documented, nor statically enforced. One might have to search developer websites to find all relevant constraints or even just run systems to find out whether configurations cause malfunctions. Indeed, even the types of fields are often not very well specified. For example, the type of Hadoop's *JVM options* parameter is *string*, but not any string will do. The lack of documentation of constraints and enforcement makes it not only easy to make configuration errors but also vastly increases search space sizes.

To reduce the search space, ConEx filters out the invalid configurations sampled by the previous phase using a configuration checker developed by Tang *et al.* [102]. It allows for the imposition of constraints on fields and sets of fields, taking advantage of Coq's expressive logic [103] to encode, and its type checker to enforce, constraints. The checker allowed us to avoid costly dynamic evaluation of about 8% of all of the configurations that ConEx sampled in the previous step. We have a different focus with Sayyad *et al.*'s work [104]. They focus on finding valid software product line configurations where correctness is the primary concern. This work focuses on finding configurations that can yield high performance while running guest jobs.

5.3 Configuration Validity Checking

Configurations are collections of parameter values that can be set by end-users to specialize and optimize system functions, performance, and other properties for particular uses or environments. Configurability enables the production of commodity software and software-intensive systems that can be used for diverse purposes.

Selecting configurations is a fraught exercise. Even individual components can have hundreds of configuration parameters. Systems of systems can have orders of magnitude more. Configurations are also often under-specified, as manifested in the use of loose *machine-level* types (e.g., *integer, string*), for configuration parameters (or *fields*), and in the incomplete and imprecise specification of constraints on and across fields. These issues often make it unclear what values

parameters can reasonably have, what they mean precisely, how to set them to obtain desired system properties (e.g., performance, security), and how not to set them to avoid comprising system properties.

The complexity, inadequate specification, and opaque meanings of configurations risks the use of bad configurations and vastly enlarges the configuration spaces that configuration engineers and auto-tuners must explore. We propose to address such problems with *interpreted formalisms for configurations*.

Earlier work by Xiang, Knight and Sullivan [105, 106] identified a lack of explicit, checkable interpretations for code as posing risks to cyber-physical system dependability. They proposed *interpreted formalisms* as a solution. An interpreted formalism augments code with an explicit structure—an *interpretation*, mapped to the code—that imposes *real-world types* on, and further explicates the intended meanings of, code elements, both to aid human understanding and to enable automated checking of code for consistency with real world constraints.

An interpreted formalism is a $(code, interpretation)$ pair. It can be used to check that machine-level values can be lifted to values of real-world types. Such types can *extend* and further *constrain* machine-type values (e.g., with units, limiting *integer* values to *positive* values, possibly with with additional range restrictions, etc.). Xiang et al. [106] demonstrated the efficacy of interpreted formalisms for finding bugs in Java programs for cyber-physical systems.

The problems we have identified with *configurations* are analogous to those with code. We introduce *interpreted formalisms for configurations* as a solution. We augment parameters and whole configurations with interpretations to explicate intended meanings and enable checking of configurations against real-world constraints. We specify real-world types as what amount to dependent pairs in Coq. Values of real-world types combine machine-level values lifted to values of Coq types, with proofs of additionally specified properties of these lifted values. Real-world type checking involves lifting followed by automated construction of proofs. Real-world type errors are detected if either lifted values or constructed proof objects fail to type check in Coq.

As evidence of the feasibility, utility, and conceptual clarity afforded by our approach, we present an interpretation for Apache Hadoop [1] configurations, including real-world types based on constraints mined from Hadoop documentation. The main contributions of this section of this chapter can be summarized as follows:

- We show that formally specified, fully automated, efficient real-world type checking can be provided for system configurations
- We show that real-world type checking can find previously unrecognized errors in Hadoop configurations
- We show that filtering malformed configurations can significantly improvement search efficiency
- We show that Coq's dependent type theory and module system support clear, practical, and flexible specification of interpreted formalisms for configurations
- We establish foundations for real-world type systems grounded in type theory

In recent work [105, 106], Xiang, Knight, and Sullivan identified two major shortcomings in today's software practice. First, software engineers tend to represent properties of real-world phenomena as values of—and in procedures that operate on values of—*under-constrained machine types*. As one example, an altitude relative to ground in meters might be represented only by a value of the machine type, *integer*, perhaps with a name such as *alt* and a comment, *altitude in meters relative to the ground*. The formal type is under-specified in that it permits values, such as -1, that are meaningless in the real world.

The second, closely related, problem is that the intended *interpretations* of code are not specified in a form that enables sufficient automated checking of consistency of code with the real world. Machine-level values and operations are permitted that have no real-world meaning. There is usually nothing to prevent a program from adding an integer (in meters) to an integer (in feet), for example.

Similar issues involve frames of reference, staleness of sensor data, measurement error, possibilities for erroneous data from failed sensors, etc.

In order to address these problems, Xiang et al. proposed the concept of the interpreted formalism based on real-world types. In contrast to the current practice, the *real-world type* assigned to *alt* might be *non-negative real integer expressed in meters above ground level (AGL)*. The real-world types constrains the value and adds units and a frame of reference. Real-world type systems limit *machine* values to values that are meaningful in the real world while extending them with information critical to the full specification and automated checking of their intended interpretations. In addition to real-world types, an interpretation can include information such as references to relevant standards, expository prose, etc., to further clarify the intended meanings of machine-typed values.

The present work emerged from an effort in combinatorial optimization of Hadoop performance through novel meta-heuristic searches for high-performing configurations. We found that the *machine types* of Hadoop configuration parameters (e.g., integer, string, float), and thus of configurations, were often under-constrained, that their intended interpretations were often unclear, and that Hadoop was without mechanisms for checking the values of parameters with real-world constraints. Many fields are documented as being of type integer, for example, even in cases where not any integer will do. We also found some Hadoop documentation to be erroneous. Hadoop's Wiki page[1] cites *io.buffer.size* as a configuration field name, but there is no such field. It appears that *io.file.buffer.size* was meant. Among other harms, under-specification enlarges search spaces to include configurations that violate known but unchecked real-world constraints.

5.3.1 Type Checker Design

To address the problems that flow from under-constrained configurations with poorly specified interpretations, we introduce *interpreted formalisms based on real-world types for configurations*. We first describe how we formalize real-world

[1] `https://wiki.apache.org/hadoop/HowManyMapsAndReduces`

types and lift machine-typed field and configuration values to real-world type checked values. Then we present an example using this mechanism to produce an interpretation for and to type check a Hadoop configuration.

Extending Configurations with Real-World Types

Configurations, which are collections of constant definitions, are simpler than imperative code. There are usually no assignments to mutable memory, function calls, pointers, sub-typing, etc. Their simplicity has enabled us to clarify our understanding of interpreted formalisms based on real-world types. We formalize a real-world type as a dependent pair type, (b_r, p_r), where b_r is what we have called a *base type* (such as *positive* in Coq), and where p_r is an additional property of values of this type—in Coq, a function from values to propositions about them—such as the property of being divisible by the hardware page size on a given machine.

Binding a real-world type to a parameter, p, with a machine value v_m (such as 65536) of machine type t_m (such as integer), involves the *lifting* of v_m to a corresponding *putative* (not yet fully checked) real-world value, v_r (such as 65536%positive), of type b_r (here *positive*), followed by the construction, *if possible*, of a proof, c_r, that this particular putative real-world value, v_r, has the additional property p_r (e.g., that *65536%positive mod 4096%Z = 0%Z)*. If a proof, c_r, can be constructed, then the dependent pair, (v_r, c_r) can be constructed, and the real-world type of the *machine* value, v_m, is thereby proved.

The lift-and-prove operation is essentially a partial function. A machine value v_m *real-world type checks* when it has an image under this function. In further detail, this function takes a given machine term, $(v_m : t_m)$—read as *machine value v_m of machine type t_m*—to a real-world term, $(v_r : b_r, c_r : p_r v_r)$—read as *the dependent pair comprising real-world value v_r of (Coq) base type, b_r, along with proof, c_r, of the proposition, $(p_r\ v_r)$, that certifies that v_r has property p_r*. Here c_r is a proof term (a value) for the proposition (a type) about v_r to which the Coq property p_r (a function) maps v_r. The lift-and-prove function is not defined for v_m if either (1) there is no v_r to which v_m can be lifted, or (2)

no proof, c_r, can be constructed to certify that v_r has the additional property, p_r.

The lifting of a machine value to a putative real-world value generally adds information that is known to the engineer but not explicit in the machine value or type. This additional information is vital for real-world type checking. The addition of constraints on permitted machine values is one example. Another would be that lifting adds information about the physical units in which a machine value is expressed, to enable checking of consistent use of units when machine values are combined. Simple machine types are thus generally lifted to more complex "base" types in Coq, to provide room for this added information. For example, we lift machine-level strings representing Hadoop JVM options (such as "-Xms1024m -Xmx4096m") to values of record types in Coq with fields of Coq type *positive* for the numerical values of the initial and maximum virtual machine stack sizes, explicit units (e.g., m for megabytes), and a constraint that the initial value not exceed the maximum value. The lifting operation itself can add and check constraints. For example, attempting to lift the machine-level integer value, -1, to a value of the Coq base type *positive* will fail to type check, irrespective of any additional property of the base-type value that would have to be checked had the lifting succeeded.

Working with Hadoop

An explicit interpretation when paired with a Hadoop machine-level configuration constitutes an interpreted formalism pair. Our interpreted formalisms precisely specify (1) the previously undocumented parameterization of configurations by *external platform characteristics*, such as the number of hardware CPUs, involved in constraints on the values of Hadoop parameters; (2) units for all relevant parameters, establishing a pattern if augmenting machine types with additional information such as units, frames of reference, etc; (3) all constraints ascertained from both official documentation and other trusted sources, expressed using a combination of (a) base types, such as *positive*, that can be more restrictive than

the underlying machine types, and (b) pairing of these lifted values with proofs of additional, declaratively specified properties.

Coq provides very expressive means for documenting properties (constraints), and powerful facilities for automating much (and in our work to date, all) of the verification of values against such constraints. It also provides trustworthy strong and static verification that all constraints are satisfied, via its foundational type checker. As an example, Hadoop informally documents but does not enforce a constraint that a certain field should have a value that is a multiple of the platform-specific hardware page size. Our interpreted formalism quickly reveals violations of this constraint in failures to generate required proofs. Use cases for such work include (1) automated real-world type checking of configurations, (2) using such type checking to reject mechanically generated, inconsistent configurations prior to costly dynamic profiling, (3) providing a formal specification of the constraints to be satisfied by a future, envisioned, constraint-driven generator of candidate configurations, e.g., using a separate SMT solver, (4) supporting the development of a human-facing interface for improved understanding of complex configurations, which will be critical for human-in-the-loop configuration search/tuning, and (5) for generation of good configurations for use in testing, and of counter-examples for use in fuzz testing. We have already developed (1) through (3) in this chapter, with (4) and (5) left for future work. We are also exploring applications of these ideas to configurations for complex, safety- and security-critical systems, including industrial robots.

Coq Implementation

This section presents the details of our Coq implementation of real-world types and type checker for Hadoop configuration.

Defined Coq Types

We begin by instantiating a record type whose fields represent *environment* parameters: parameters not defined as part of Hadoop configurations but that

are implicated in constraints on configurations values. For example, the number of CPU cores that MapReduce jobs are permitted to use must not exceed the number of CPUs made available to Hadoop by the hardware and surrounding system, an environment parameter. The following code presents the Coq record *type*. The fields reflect all external parameters that we know to be involved in constraints on the subset of performance-related Hadoop parameters that we have modeled. We elide the imports of libraries for the Coq types used in this code. Details can be found in our GitHub repository at `https://github.com/ChongTang/SoS_Coq`.

```
Record Env := mk_env {
    env_phys_CPU_cores: positive;
    env_virt_CPU_cores: positive;
    env_phys_mem_mb: positive;
    env_virt_mem_mb: positive;
    env_hw_page_size: positive;
    env_max_file_desc: positive;
    env_max_threads: positive;
    env_comp_codecs: list string }.
```

We instantiate a record of this type to specify a particular operating environment. In the following code, for example, the list of class names for codecs available in the Java search path on the given platform is encoded as a list of strings. This will enable us later to define and enforce a constraint that a string-valued Hadoop parameter listing codec class names include only values in this list. This environment description record is visible in the parts of our code where one defines constraints on Hadoop field values and whole configurations.

```
Definition myEnv:Env := mk_env
    14%positive
    28%positive
    32768%positive
    32768%positive
    4096%positive
    3000%positive
    500%positive
    ("org.apache.hadoop.io.compress.DefaultCodec"::...::nil).
```

Next, we formalize real-world types in Coq. As we stated in section 5.3.1, a real-world type is essentially a dependent pair type, combining a value and a proof of a property about it. We define a type, *RTipe*, the values of which designate the Coq base types for real-world types. These base types are the types to which we will attempt to lift values of concrete machine types extracted from Hadoop configuration files and objects. The mapping from these *RTipe* values to actual Coq types is given by a function, *typeOfTipe*, elided here. This mechanism allows us to write code that makes decisions based on real-world types, as one cannot *match* on actual types in Coq. Arbitrarily complex Coq types can be used as base types. We use Coq-library-provided string, integer (Z), positive integer (positive), non-negative integer (N), floating point (float), and boolean (bool) types, along with a record type that we defined to represent values of Java VM options, and an *option positive* type for fields that require either a positive integer value or a special integer, typically -1 or 0, to indicate that an exceptional behavior is required. We could, if necessary, use records that also encode units, frames of reference, and other information critical to explicating and checking real-world types.

```
Inductive RTipe := rTipe_Z | rTipe_pos | rTipe_N | rTipe_string |
    rTipe_bool | rTipe_JavaOpts | rTipe_float | rTipe_option_pos.
```

The core of our design is the parameterized type, *Field*, an instance of which

is used to represent a *certified* Hadoop field holding a lifted value for which a requisite proof of the associated property has been provided. The default property imposes no additional constraints. The *Field* type has two parameters. The first specifies the *RTipe* of the base type to which a machine value for this field will be lifted. The second specifies the additional property that must hold for any provided value of that base type. A property is represented in Coq as a function from a value of such a type to a proposition about that value. A *Field* type thus amounts to a dependent pair type with a few extra fields: (1) *field_id*: the string name of the Hadoop field (such as *"io.file.buffer.size"*); (2) *field_final*: a boolean value indicating whether the field is *final* in the sense of Hadoop, i.e., that the value can't be overridden; (3) *field_value*: a value of the Coq base type specified by the *RTipe*; and (4) *field_proof*: a proof that that particular value satisfies the additionally specified property.

```
Inductive Field (tipe: RTipe) (property: (typeOfTipe tipe) → Prop) :=
  mk_field {
    field_id: string;
    field_final: bool;
    field_value: (typeOfTipe tipe);
    field_proof: property field_value; }.
```

Generate Coq Modules from Configuration

Our next step is to generate one Coq *module* for each Hadoop configuration field to be formalized. Each such module will export the parameterized *Field* type for the corresponding Hadoop field, a function for creating values of this type, and functions for getting values of the fields of these *Field* objects, including the Coq base value in a given *Field* instance.

We use the Coq module system to generate these modules. To do this, we first define a Coq *module type* (a kind of abstract interface) named *Field_ModuleType*. The Coq code is elided here. It specifies what field-specific information has to be provided for each field to generate the required module. We then generate one

intermediate module, conforming to this interface, for each Hadoop field to be formalized. We automate this process with a Python script. Each such module provides field-specific data: the Hadoop field name (a string), its *RTipe* and thus indirectly its Coq base type, the additional property that the value of this type must satisfy, measurement units (if any), and two strings, one for a natural language explication of the meaning of the field, and another for guidance on how to set the field value. Our Python script maps machine types to *RTipe* specifications in each such module, stubbing out the additional properties to be *fun value* ⇒ *True* and stubbing out the remaining fields, which we don't yet use, to be empty strings. We hand-edit these modules to specify any more restrictive field-level constraints (e.g., here that the $io.file.buffer.size$ value should be divisible by the hardware page size). Here is an example.

```
Module io_file_buffer_size_desc <: Field_ModuleType.
  Definition fName := "io.file.buffer.size".
  Definition rTipe := rTipe_pos.
  Definition rProperty := fun value: positive ⇒
((Zpos value) mod (Zpos (myEnv.(env_hw_page_size)))) = 0%Z.
  Definition fUnit := "".
  Definition fInterp := "".
  Definition fAdvice := "".
End io_file_buffer_size_desc.
```

Finally, we run each such module through a *module functor* to produce the required module for the given field (details elided). These modules provide the $Field$ types and associated functions used in constructing and accessing values encoded in $Field$ objects. Details can be found in the source code.

Having formalized Hadoop fields, we now formalize the types of *multi-field configurations* as record types with fields whose types are the $Field$ types exported by these per-field modules. The following code, for example, formalizes Hadoop's *core-config* configuration. Each field has the same name as its corresponding Hadoop field except that dots are replaced by underscores due to Coq naming

conventions. The type of each field is specified to be the *Field* type exported by the corresponding field module. A value of this type will then represent an actual, concrete, certified Hadoop *core* configuration object.

```
Record CoreConfig := mk_core_config {
    io_file_buffer_size: io_file_buffer_size.ftype;
    io_map_index_interval: io_map_index_interval.ftype;
    io_map_index_skip: io_map_index_skip.ftype;
    io_seqfile_compress_blocksize: io_seqfile_compress_blocksize.ftype;
    io_seqfile_sorter_recordlimit: io_seqfile_sorter_recordlimit.ftype;
    ipc_maximum_data_length: ipc_maximum_data_length.ftype}.
```

Whereas we specify constraints on individual field values within *Field* objects, we specify constraints on whole configurations by including in their type definitions extra fields of *propositional types*. As an example, at the end of MapReduce configuration type we specify a multi-field constraint saying that the maximum size of the input data chunk must be greater than the minimum size. In this way, we have fully formalized the real-world types of configurations for Hadoop's core, HDFS, Yarn, and Map-Reduce components and of overall Hadoop configurations. Here's an example of the kind of constraint we can specify for configuration objects.

```
maxsplit_lt_minsplit:
    Z.gt (Zpos (mapreduce_input_fileinputformat_split_maxsize.value
                mapreduce_input_fileinputformat_split_maxsize))
         (Z.of_N (mapreduce_input_fileinputformat_split_minsize.value
                  mapreduce_input_fileinputformat_split_minsize))
```

5.3.2 Initialize and Check Configuration

We now use a Python script to lift Hadoop configurations to values of Coq configurations types to type check them. Lifted configurations look much like real configuration files. See the following example, in which we use the *mk_yarn_config*

constructor to instantiate a Coq configuration object, a_yarn_config, of type $YarnConfig$. For each field, we generate a call to the *mk* function from the per-field $Field$ module to instantiate a $Field$ object of the requisite type, providing the required values for its components: (1) a boolean value specifying whether the value is final or not (the false's); (2) a field value, now of a value of the required Coq base type; and (3) a proof object to prove that the value of the field satisfies the properties specified for that value, but using an underscore as a *hole* for a proof to be constructed using Coq tactics. We provide additional proof objects, again as *holes*, for the cross-field constraints (elided here). The whole definition is wrapped in a Coq *unshelve refine* tactic, with a tactic-based proof building script at the end that fills in the required proof objects if it's possible to construct them.

```
Definition a_yarn_config: YarnConfig.
Proof.
unshelve refine (
    mk_yarn_config
      (yarn_nodemanager_container__manager_thread__count.mk
            false 20%positive _ )
      ...
      (yarn_sharedcache_admin_thread__count.mk
            false 1%positive _ )
      ... );
try (exact I); try compute; try reflexivity; auto.
Qed.
```

We specify a real-world type for an entire Hadoop configuration as a Record whose fields are values of the real-world types of the four Hadoop subsystems. We anticipate that the methods developed here can be adapted to deeply hierarchically structured configurations for large and complex systems.

```
Record HadoopConfig := mk_hadoop_config {
    yarn_config: YarnConfig;
    mapred_config: MapRedConfig;
    core_config: CoreConfig;
    hdfs_config: HDFSConfig}.
```

Given a complete, machine-level Hadoop configuration, with core, map-reduce, Yarn, and HDFS sub-configurations, our Python script lifts it to a corresponding value of this $HadoopConfig$ type. In this way, machine-type field and whole configuration values that encode real-world concepts get converted to values of real-world types that make their full real-world meanings explicit and subject to mechanical checking for real-world consistency.

5.3.3 Checker Evaluation

We now consider the extent to which this work makes the contributions claimed in the introduction.

An Advance in Real-World Type Systems

This work has demonstrated the feasibility and effectiveness of constructing interpreted formalisms based on real-world types for complex configurations. It has shown how Coq's type system can be used to define real-world types that clearly express the essential properties of otherwise inadequately typed machine values. As an example, Hadoop encodes values of what are essentially *option positive* real-world types as mere *integers*, with either 0 or -1 (inconsistently) representing *None*. Coq's parameterized algebraic data types (such as *option* T), and its *propositions as types* paradigm, enable the highly expressive representation and trustworthy checking of an unlimited range of real-world types. Representing real world types as Coq types rather than as the simple and somewhat inflexible record types in the original work of Xiang et al. represents a significant advance over the prior state of the art in real-world type systems.

Detecting Real-World Errors in Configurations

One of the main purposes of a real-world type system is to reveal inconsistencies in software that elude machine-level type systems. Our case study demonstrates the potential for real-world type systems to find inconsistencies in configurations. The context of this chapter is a project on meta-heuristic search through spaces of configurations. Our work to date generates Hadoop configurations in spaces spanned by the specifications of a few machine-typed values to be considered for each Hadoop parameter. Unfortunately, not every combination of machine-type values make sense in the real world. Interposing our real-world type checker between our configuration generator and the costly experimental profiling operation allows us to greatly improve search performance by eliminating many configurations from consideration before subjecting them to costly experimental evaluation. Here are a few concrete examples.

As one example, the machine type of *mapreduce.jobtracker.maxtasks.perjob* is integer, where a positive value imposes a resource limit and -1 means no limit. Our generator was programmed to allow this field value to vary between -1 and 4 based on the machine type of the field. A problem is that a value of 0 actually makes no sense for this field, as that would indicate that the maximum number of tasks that can be allocated to a given job is zero. Adding a constraint that the field not be 0, which we did by lifting the field to the real-world *option positive* type, eliminated many nonsensical configurations from consideration. Lifting 0 to $Some 0\%positive$ yields a Coq term that simply doesn't type check.

Using properties to further constraint lifted terms of Coq base types also revealed real-world inconsistencies. The formula $min(min\_splitsize, min(blocksize, max\_splitsize)$, for example, is used to compute the chunk size in Hadoop, where *bloacksize* is the size of a data block in HDFS. If the $min\_splitsize$ is greater than $max\_splitsize$, the final chunk size will be the smaller of the values of *blocksize* and $max\_splitsize$, which is semantically wrong. Although a MapReduce job won't fail because of this error, it will behave in unexpected ways. Our type checker finds violations of this constraint.

Another cross-field constraint violation that our type checker found to our surprise had to do with a set of four constraints about Hadoop's *uber mode*. The constraints are documented in Hadoop's official documentation [2]. They say that if users enable *uber mode*, the CPU and memory resources of *map* and *reduce* tasks must be less than those of the application master.

It is not surprising that adding constraints invalidates some, or even many, configurations. The concept of constraint-driven design space exploration isn't new. A more interesting implication is that what we should be doing is to base our configuration generator on the real-world types of configurations rather than on their machine types! Consider again the *mapreduce.jobtracker.maxtasks.perjob* field. A -1 value indicates not just another numerical limit, but rather is a flag indicating "no limit is imposed." A generator should treat "no limit" as fundamentally different than 1 or 2 or 3. A multi-level exploration strategy is then called for—either no limit or one of a range of numerical values. Proper consideration of the *real-world* types of field can inform meta-heuristic search strategies, a point we plan to pursue further in future work.

Net Improvement in Meta-Heuristic Search Performance

To produce a data point on how filtering constraint-violating configurations can improve search performance, we used our real-world type checker to type-check $5,000$ randomly generated configurations, of the kind we generate and test in our search methods. $1,293$ were invalid. One invocation of our runtime Hadoop performance profiling operation takes about 30 seconds. We run each job 3 times to obtain an average performance measurement. The saved time is the difference between the time needed to dynamically evaluate $1,293$ configurations and the time needed to type-check $5,000$ configurations. The time to dynamically profile Hadoop running under $1,293$ configurations was about $1293 * 30 * 3 = 116370$ seconds. Each type check takes about 0.63 seconds. The total time to check $5,000$ configurations was thus about $5000 * 0.63 = 3150$ seconds. The saved time was $116370 - 3150 = 113220$ seconds out of a total time of $5000 * 30 * 3 = 450000$ seconds.

[2] `https://hadoop.apache.org/docs/r2.7.4/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml`

The saved time is about 113220/450000 ≈ 0.25, or 25% of the total search time. Specifying and checking informally and often incompletely documented constraints on configurations can clearly reduce search spaces and improve search efficiency significantly.

A Flexible Real-World Type System for Configurations

Our real-world type system for Hadoop configurations has been easy to use. We wrote a Python script to (1) instantiate *Field* meta-data modules for each Hadoop configuration field, based on a spreadsheet, in which for each field we entered information about field name, machine type, Coq base type, and natural language explications of intended interpretations along with guidance for configuration engineers, and (2) generate all associated configuration type specifications. Once this code is synthesized, the remaining tasks are to edit additional properties in-by-hand and to create and check configuration objects, which we do by automatically running the *coqc* command-line Coq type checker on the generated files. It is easy to add and extend real world types to the system: on the order of an hour of work in our experience.

Precise Formal Specification of Configuration Spaces

Our specification of the real-world type of a Hadoop configuration provides an authoritative formal specification of this configuration space, and as a template for specifications of other such configuration spaces. It precisely specifies of the set of all and only valid Hadoop configurations, limited here to a subset of about 100 performance-related fields. In particular, we formalize configuration spaces as types in the constructive logic of Coq. This work enables a precise specification the optimization problem that motivated this work: find *argmin (c: HadoopConfig) runtime(b,c),* where c encodes a configuration in a particular context, b is a benchmark Hadoop job, and $HadoopConfig$ is the real-world type of Hadoop configurations. Optimizing system quality attributes by searching over dependently typed representations thus emerges as a fundamental mathematical problem formulation that seems worthy of further consideration.

5.4 Experimental Design

We implement ConEx with about 4000 lines of Python code. The tool is available in a public Github repository: `https://github.com/ChongTang/sysopt`.

5.4.1 Study Subject

We developed ConEx by focusing on Hadoop [87] and used Spark [107] for validating the approach. Hadoop and Spark are the two most popular big data framework today. The main difference between them is that Spark uses memory as much as possible in lieu of mass storage to reduce execution times.

Table 5.1 summarizes the parameters and their types that we have studied for Hadoop $v2.7.4$ and Spark $v2.2.0$. Hadoop has 901 parameters across four sub-systems (CORE, HDFS, MAPREDUCE, and YARN). After Phase-I, we identified 44 parameters relevant to performance. Similarly, 27 out of 212 Spark parameters are selected. These parameters, in total, produces $3 * 10^{28}$ and $4 * 10^{16}$ configurations respectively. Such configuration space is several orders of magnitude larger and complex than the configuration space studied before [11]. To further reduce the search space, for numeric parameters, we select $\pm 10\%$ values around the default and all valid values for boolean, string, and categorical types.

To evaluate ConEx, we select big-data jobs from HiBench [108], a popular benchmark suite for big data framework evaluation. It provides benchmark jobs for both Hadoop and Spark. For Hadoop, we selected five jobs: WordCount, Sort, TeraSort, PageRank, and NutchIndex from the Micro and Websearch categories. These jobs only need a core Hadoop system to execute as opposed to other categories that require Hive or Spark. For Spark, we selected five Spark jobs: WordCount, Sort, TeraSort, RF, and SVD. HiBench has six different sizes of input workload, from “tiny” to “Bigdata”. For our experiments, we used “small", “large”, and “huge” data inputs. Table 5.5 shows the CPU times taken by the Hadoop jobs running with default configurations. As the running time of the

larger jobs much higher compared to the small jobs, we experimented with smaller jobs and validated with the larger workloads (*Scale-up* hypothesis).

We conducted our experiments in our in-house Hadoop and Spark clusters having one master node and four slave nodes. Each node has a Intel(R) Xeon(R) E5-2660 CPU and 32GB memory. We assigned 20GB for Hadoop on each node in our experiments. We also made sure that no other programs were running except core Linux OS processes.

5.4.2 Job Classification

To reduce sampling cost, we proposed our *scale-out* hypothesis, *i.e.* configurations found to be good for one kind of job might also work well for other, similar types of jobs. To test the hypothesis we needed a way to cluster jobs by their resource usage. HiBench classifies jobs by their business purposes (*e.g.* web page indexing and ranking related jobs fall in the *Websearch* category). However, such a classification does not necessarily reflect their true resource usage. We thus developed an approach to clustering jobs by profiling their run-time behaviors based on system call traces. Similar approaches have been widely used in the security community [109–111].

A Unix command line tool `strace` captures system call traces of a process, typically running on a single machine, and logs how the process has used system resources. We configure Hadoop and Spark to run on the single-node model; here, a task runs as a single Java process. We then collect call traces of all studied jobs.

Based on the system call traces, we represent each job by four-tuples: $< A, B, C, D >$, where A: call sequence, B: a set of unique string and categorical arguments across all system calls, C: term frequencies of string and categorical arguments captured per system call, and D: the mean value of the numerical arguments per system call. Table 5.2 shows an example tuple.

Table 5.2: Example Tuple representing resource usage of a job

Example System Call Sequence	$foo(1, "b"), bar("b", True)$, $foo(2, "b"), foo(3, "c")$
A	$\{foo, bar, foo, foo\}$
B	$\{foo : ("b", "c"), bar : ("b")\}$
C	$\{foo : ["b" : 0.66, "c" : 0.33], bar : ["b" : 1.0]\}$
D	$\{foo : ["1^{st} arg" : 2.0]\}$

Table 5.3: Performance improvement for Hadoop jobs from three sampling strategies

	Exploration Phase	**Scale-up Phase**	
Job	**Small**	**Large**	**Huge**
	EMCMC		
WordCount	12.5%	15.6%	21.5%
Sort	72.1%	7.7%	15.8%
TeraSort	27.4%	16.1%	18.3%
PageRank	32.7%	44.7%	25.2%
NutchIndex	7.1%	18.7%	14.2%
	Genetic Algorithm		
WordCount	6.5%	9.8%	11.6%
Sort	70.6%	11.4%	10.3%
TeraSort	21.6%	17.2%	15.9%
PageRank	46.4%	17.5%	21.2%
NutchIndex	5.7%	11.5%	13.4%
	Random Sampling		
WordCount	5.6%	9.7%	14.6%
Sort	60.6%	5.3%	13.5%
TeraSort	11.6%	8.6%	11.1%
PageRank	32.9%	12.0%	11.2%
NutchIndex	10.4%	14.4%	12.0%

To compute similarity between two jobs, we calculate the similarities between each tuple-element separately, which contributes equally to the overall similarity estimation. For tuple elements A, we use pattern matching—we slice the call sequences and compute the similarity between them. To find similarities between two B elements, we compute the Jaccard Index, which is a common approach to compute the similarity of two sets. For C elements, we compute the average difference of each term frequency. Finally, for D elements, we compute the similarity of mean value of numerical arguments as $1 - abs(mean1, mean2)/max(mean1, mean2)$. We take the average value of these four scores as the final similarity score between two jobs. We consider two

jobs is *similar*, if their similarity score is above 0.77 (*i.e.* from third quartile (Q3) of all the similarity scores).

5.4.3 Comparing with Baselines

We compare ConEx's performance with three baselines: (i) Random sampling, (ii) Genetic algorithm based evolutionary sampling, and (iii) Learning-based model. The first two evaluate whether EMCMC is a good sampling strategy over other sampling strategies. The last baseline evaluates the choice of meta-heuristic search over popular learning-based approach. In particular, we compare ConEx with Nair *et al.*'s ranking based approach [112] as they showed that their rank-based approach requires fewer samples (an important requirement for big-data systems) over other residual-based approaches.

5.5 Experimental Results

We will now discuss the way we evaluate ConEx and the results. We developed ConEx using Hadoop jobs and present the experimental results. To test the generalizability of our approach, we have also run and report data from experiments using the Spark framework. We start our experiments with the basic question:

RQ1. Can ConEx find better configurations than the baseline configuration within a given computation budget?

We investigate this RQ by exploring the configuration space using small workloads. First, ConEx runs the benchmark jobs by setting the workload size "Small" in the HiBench configuration file. HiBench generates a detailed report after each run, with diverse performance information including CPU time. For Spark jobs, we use larger dataset instead of the small workload because spark jobs run very fast under small workload—it is difficult to observe any meaningful performance gain. Thus, we use a larger workload for exploration, which is about the same

size as "large" workload defined in HiBench, tailored to take about 30 seconds per run, making the cost of exploration comparable to that of Hadoop.

Each execution in HiBench contains two steps: data preparation and job execution. However, we manually prepare the data so that each job under test can be run with the same workload. Hence, we modify HiBench to bypass its first step and run jobs with only user-provided data. At the end of each exploration, some good configurations are output along with their performance data, as well as how they are compared *w.r.t.* the baseline configuration (default value in our case). Thus, if the performances for the default and the best configurations are $perf_d$ and $perf_b$, the performance improvement will be $\frac{perf_d - perf_b}{perf_d}$.

Table 5.3 shows the results: configurations found by ConEx with EMCMC approach achieve 7% to 72% better performances than default configurations for five Hadoop jobs. For Spark jobs, ConEx finds 2.7% to 40.4% performance improvements for all five jobs (see "Exploration" column of Table 5.4). The highest improvement is for TeraSort (40.4%).

> **Result 1:** *For Hadoop and Spark jobs with small workload,* ConEx *can find configurations that produce up to 72% and 40% performance gains respectively over the default configurations.*

Even if ConEx manages to find a better configuration with small workloads, ConEx will be most effective if it can improve performance for larger workloads and thus save significant cost. This leads us to question:

RQ2. Scale-Up: Do configurations found with small workloads also produce significant improvements in performance for much larger workloads?

Here, we run the same HiBench jobs as used in RQ1 with "Large" and "Huge" inputs—10X and 100X times larger workloads respectively (*i.e.* more than 3-GB and 30-GB inputs). For a given job, we choose top 50 best performing configurations from RQ1 and profile them along with the default configuration

and record the CPU times. We then compare the performance gains *w.r.t.* the baseline performance.

Note that, here we choose the top 50 configurations found in the exploration phase (RQ1) as opposed to the best one because we found them using small workloads; the behaviors of small and larger workloads may not be exactly same. Thus, using such *multi-resolution* approach, we aim to tolerate the variation of workload size, and yet gain save full-blown exploration cost.

Table 5.3 presents the results for Hadoop jobs. For all five jobs, we see an average of 20.6% and 19% improvements under large and huge workloads. In fact, for WordCount, PageRank, and NutchIndex jobs, large workloads achieve better performance gain than the improvement under the small ones.

Table 5.4: EMCMC results for Spark jobs

Job	**Exploration**	**Scale-Up**
WordCount	2.7%	5.8%
Sort	3.3%	1.6%
TeraSort	40.4%	16.7%
RandomForest	6.4%	7.2%
SVD	0.4%	1.9%
Average	10.64%	6.7%

Similarly, for all Spark jobs we see performance improvements for all the jobs (see Table 5.4). Here, we use huge workload for scale-up evaluation, since we use large data in the exploration phase. We saw performance improvements for all five jobs, ranging from 1.6% to 16.7%. However, for Sort and SVD, there are limited improvements: 1.6% and 1.9% respectively. Spark jobs run very fast compared to Hadoop jobs. For example, the Thus, although we have scaled up the workload 10 times, it becomes difficult to achieve significant gain. Additional research is needed to better understand the scale-up potential of Spark jobs. Nevertheless, we saw an average performance improvement of 6.7%, which we believe is non-trivial at this scale.

Sensitivity Analysis. Here we study how sensitive the performance gain is *w.r.t.* each configuration parameter. From the best-found configuration, we

Table 5.5: CPU time (secs) of default configuration for each job under three data inputs

	Small	Large	Huge	#EXP
WordCount	166.2	862.6	9367.1	#3241 (Gen17)
Sort	133.4	869.8	9891.7	#3318 (Gen17)
Terasort	115.7	1056.6	8751.6	#2876 (Gen15)
Pagerank	300.0	5657.2	13096.1	#3177 (Gen16)
NutchIndex	477.9	6596.5	11215.7	#4685 (Gen24)

set the value of each parameter back to its default value and check how much the performance has changed. For example, say $perf_{def}$ and $perf_{best}$ are the default and best performances (*i.e.* CPU times) obtained by ConEx for a job. Then, the performance improvement *w.r.t.* the default configuration is $\Delta_{best} = \frac{perf_{def} - perf_{best}}{perf_{def}}$. Note that, this is the best gain observed by ConEx. Next, to measure how sensitive the gain is *w.r.t.* a parameter c_i, we set c_i's value back to default without changing the other parameter values from the best configurations. We measure the new performance *w.r.t.* to the default; Thus, $\Delta_i = (perf_{def} - perf_i)/perf_{def}$. Then the sensitivity of parameter c_i is the difference of performance improvement: $sensitivity_i = \Delta_{best} - \Delta_i$.

We conduct this analysis for all the parameters one by one for the Hadoop jobs with "huge" workload. Table 5.6 shows the results. The second row is the overall performance gain. [3] It shows that performance improvement is sensitive to only few parameters. However, no single parameter is responsible for most of the improvement. Instead, the data seem to indicate that the influences of individual parameters are limited and that overall improvements come from combinations of, or interactions among, different parameters. These results suggest that, at least for Hadoop, higher-order interactions are present in the objective function and that these will need to be addressed by algorithms that seek high performing configurations.

Cost saving. The first three columns in Table 5.5 show the total execution time (in seconds) across the master-slave nodes for each Hadoop job under three workload sizes on our test platform. For example, *WordCount* under small

[3] We used a different cluster to do sensitivity analysis. So the overall performance could be slightly different from those in Table 5.3.

Table 5.6: Most Influential Parameters for Hadoop Jobs

Jobs **Total Gain**	**Ranked Parameters**
WordCount **22.34%**	mapreduce.map.memory.mb: 12.61% mapreduce.map.sort.spill.percent: 3.76% mapreduce.reduce.input.buffer.percent: -0.48% mapreduce.job.max.split.locations: -0.67% yarn.app.mapreduce.am.resource.mb: -1.54%
Sort **19.77%**	mapreduce.reduce.input.buffer.percent: 12.13% mapreduce.map.memory.mb: 4.29% io.seqfile.compress.blocksize: 1.83% io.file.buffer.size: 1.58% mapreduce.map.java.opts: 1.11%
TeraSort **18.45%**	mapreduce.map.java.opts: 8.65% mapreduce.reduce.input.buffer.percent: 7.22% mapreduce.task.io.sort.mb: 2.84% mapreduce.reduce.memory.mb: 1.75% io.seqfile.compress.blocksize: 1.58%
PageRank **19.56%**	mapreduce.map.memory.mb: 10.82% mapreduce.reduce.input.buffer.percent: 5.38% mapreduce.map.java.opts: 3.10% mapreduce.task.io.sort.mb: 2.38% mapreduce.reduce.memory.mb: 1.95%
NutchIndex **19.04%**	mapreduce.map.java.opts: 9.44% mapreduce.task.io.sort.mb: 6.19% mapreduce.reduce.memory.mb: 4.92% mapreduce.map.memory.mb: 1.83% yarn.resourcemanager.scheduler.class: 0.39%

workload takes about 33 seconds per cluster (166.2/5, where 5 is the number of nodes of our cluster). However, it takes around 1873 seconds with "huge" data, with the time difference of 1840 seconds. The last column shows the number of dynamic evaluations of sampled configurations before ConEx achieves the best configuration. Thus, for *WordCount* job, our step-up hypothesis saves 1656 hours ($(1873 - 33) * 3241$ seconds) to find a better configuration. In total, for all the five jobs, the scale-up hypothesis saves about $9,600$ hours or 39.6 times.

Result 2: *Our data support the scale-up hypothesis: good configurations found using small workloads as proxies produce significant performance improvement with larger workloads. Thus, it saves a significant experimental cost.*

Further exploration cost can be saved if our scale-out hypothesis holds good, *i.e.* configuration found for one kind of job, A (e.g., Word Count), will also produce performance gains for a similar kind of job, B (e.g., Sort). Thus, we ask:

RQ3. Scale-Out: Do configurations found for one kind of job produce significant performance improvements for similar types of jobs?

We test this RQ by evaluating performance gains for job B using configurations found for job A, where the similarity between A and B is measured using Section 5.4.2. Among the five Hadoop jobs, we find that *WordCount*, *Sort*, *TeraSort* are highly similar, and *PageRank* is somewhat similar to them, whereas *NutchIndex* has low similarity with this group. Table 5.7 shows the results of Scale-Out hypothesis testing. The row header (first column "Rep.Job") indicates the representative job, and the column headers represent the target jobs. The numbers indicate the performance gains achieved by a target job while running with the best configuration of Rep.Job for huge workload. The underlined numbers highlight the best performance in each column. The numbers in the diagonal show the performance achieved by a job when tested with its own best configuration (representative job=target job).

For example, Table 5.7a shows that ConEx found a configuration for *WordCount (WC)* that improves its performance by 21.5%. When the same configuration is used for the similar target jobs: *Sort*, *TeraSort*, *PageRank*, the performance gains achieved (10.7%, 28.1%, and 20.6% respectively) are close to the improvements found by their own best configurations. However, for *NutchIndex*, which is not so similar, we see a performance gain of only 7.1%, while it achieved 14.2% gain while experimenting with its own best configuration. Similar conclusions can be drawn for Spark jobs.

A surprising aspect of these results is that, in few cases, a better configuration found for one job, e.g., Nutch, yielded greater gains for another job, e.g., Sort (27.6%) than the gain achieved by its own best configuration (15.8% for Sort). We speculate that dynamic characteristics of jobs such as Nutch might have

Table 5.7: Scale-out Hypothesis Testing

(a) **Hadoop**

Rep. Job	**Similar Jobs**				**Diff. Job**
	WC	**Sort**	**TeraSort**	**PageRank**	**Nutch**
WC	*21.5%*	10.7%	28.1%	20.6%	7.1%
Sort	11.4%	*15.8%*	21.1%	18.4%	5.7%
TeraSort	10.4%	1.8%	*18.3%*	29.9%	3.8%
PageRank	20.8%	23.8%	23.4%	*25.2%*	16.8%
Nutch	12.2%	27.6%	15.5%	10.4%	*14.2%*

(b) **Spark**

Rep. Job	**Similar Jobs**				
	WC	Sort	TeraSort	RF	SVD
WC	*5.8%*	50.8%	22.8%	5.9%	1.8%
Sort	3.5%	*1.6%*	22.3%	9.9%	4.7%
TeraSort	5.1%	17.8%	*16.7%*	9.9%	3.1%
RF	4.3%	20.1%	10.0%	*7.2%*	2.5%
SVD	2.2%	23.8%	13.4%	23.4%	*1.9%*

exerted greater evolutionary pressure than those of other jobs. But we have not adequately explored the causes of these surprising results, and plan to do so in future work.

Result 3: *Our scale-out hypothesis holds well, i.e., the configuration found with a representative job can bring significant performance gain for other similar jobs.*

Finally, we evaluate ConEx *w.r.t.* the baseline approaches as described in Section 5.4.3. We present the results for only Haddop jobs here.

In particular we check:

RQ4. How does EMCMC sampling strategy perform compared to alternative sampling strategies?

Here we compare EMCMC with (i) random and (ii) genetic algorithm (GA) based evolutionary sampling strategies. A random approach samples a parameter

value from the uniform distribution, *i.e.* each value in the value range has the same probability to be selected. We have also implemented a GA based sampling strategy with the same cross-over and mutation strategies and the same fitness function as of EMCMC. For comparison, we run the baseline strategies to generate the same number of configurations and profile their performances with "Small" data sets. We then conduct the scale-out validation to evaluate the performance gain in larger workloads.

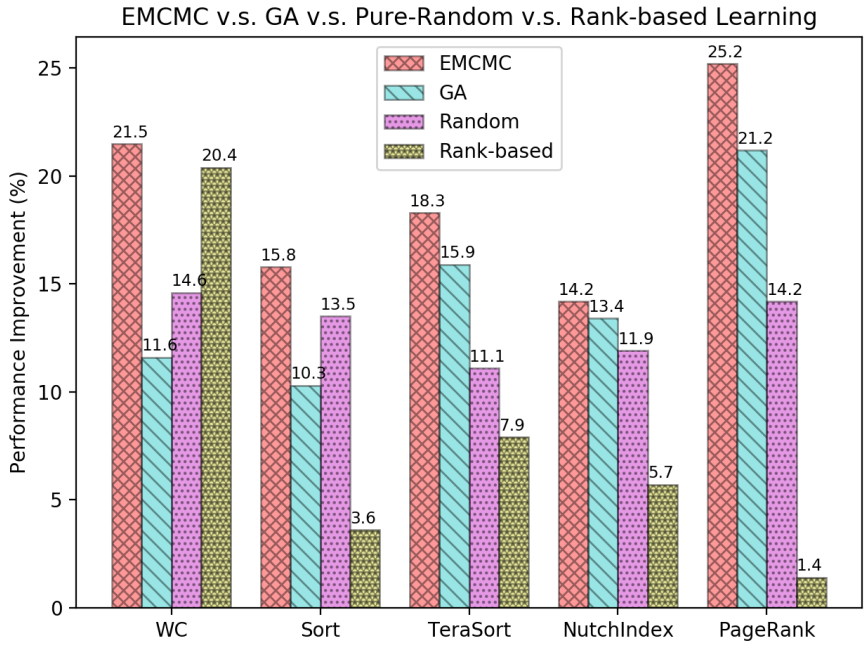

Figure 5.2: EMCMC compares with other approaches in performance improvement for Hadoop Huge Workload

Table 5.3 shows the detailed results for different workloads. Overall, for all the jobs, EMCMC based sampling performs better. Figure 5.2 pictorially represents the results for "Huge" workload. EMCMC outperforms the random strategy from 17% to 125% under Huge workload across all the studied Hadoop jobs. EMCMC based sampling strategy also achieves better gain than the genetic algorithm (GA) based evolutionary search. EMCMC performs 6% to 85% better than GA for all the five jobs. The improvement of performance of EMCMC over GA also gives us an estimate of how much the evolutionary part of EMCMC contributes to ConEx's performance.

Result 4: *EMCMC based sampling strategy outperforms random and genetic algorithm based evolutionary sampling strategies to find better performing configurations.*

RQ5. How does search-based approach perform compared to learning-based approaches?

To compare search based strategy over learning-based approaches, we choose the state-of-the-art work of Nair *et al.* [11] published in FSE 2017. They used a rank-based performance prediction model to find better configuration. The authors argue that such a model works well when the cost of sampling is high, such as ours. They show that compared to residual based approaches, their model saves a few magnitudes of sampling cost and achieves similar and even better results.

In their experiments with larger systems having millions of configurations (*e.g.* SQLite), the training pool covers 4500 configurations, including 4400 feature-wisely and pair-wisely sampled and extra 100 random configurations. We used the same approach—we randomly collected the same number of configurations as of ConEx to profile their performances (similar to RQ4) and used them as training. We directly used the model published by Nair *et al.*. Same as they did, we ran each model 20 times to get the best configurations.

For a fair comparison, following Nair et al., we evaluated both the approaches by measuring rank difference (RD) between the predicted rank of a configuration and the rank of the training data (the profiled performance in our case). Table 5.8 shows the result. Here we ran each model 1000 times to get enough data for the descriptive analysis. The results show that although the minimum RD is 0, the average and maximum *RDs* are 13.2 and 408 respectively, and the standard deviation is 24.4. It means that this model could be largely wrong when trying to find high-performing configurations.

Note that, both approaches cannot guarantee to find the best candidate. So we discuss which approach can find the best candidate from all configurations

Table 5.8: Descriptive rank differences of 1000 tests

Job	Mean	Std	Min	Max
WordCount	13.2	24.4	0	408
Sort	28.7	42.6	0	391
TeraSort	14.3	19.1	0	171
NutchIndex	16.4	24.0	0	296
PageRank	9.5	16.7	0	158

they checked. As we see from Table 5.8, although a learning-based approach can find good configurations, it cannot guarantee the resulting one is the best. In some cases, the ranking mistake could be as large as 408. On the other hand, our search-based approach can accurately find the best thanks to the dynamic evaluation and guided searching algorithms.

How much performance improvement one can gain by using Nair *et al.*'s approach? While our final goal is to improve system performance, we studied which approach can find better configurations, concerning how much performance one can gain. Suppose an engineer wants to use their approach to find a good configuration. She knows that all learning-based approaches have prediction errors. One possible way is to run such a model multiple times to rank configurations and then find the one with the best ranking in average across all tries. In this paper, we modified the tool released by Nair et al. to get the actual ranking of configurations. We run the above-described procedure 20 times and find out the configuration with the highest rank in average. The last bar in Figure 5.2 shows the performance improvement of the rank-based approach *w.r.t.* the default configuration. ConEx performs 5.4% to 338.9% better than the ranked-based approach across five Hadoop jobs.

To understand why Nair *et al.*'s approach doesn't perform well in finding good Hadoop configuration, we studied the accuracy of the trained models. In their implementation, the ranked-based model wraps a decision tree regressor as the underhood performance prediction model. We checked the R^2 scores of these regressors, and it turns out that all scores are negative for all five jobs. It means that the trained model performs arbitrarily worse. This is not surprising because Hadoop's configuration space is complex, higherarchical, and high-dimensional;

it is hard for the models to learn a function approximating such space. A neural network based regression model might work better; However, that would incur more sampling cost to gather adequate training sample.

Result 5: *Compared to Nair et al's learning-based approach, EMCMC driven search-based strategy performs better at finding configurations with higher performance gains.*

5.6 Related Work

Real-world type checking The type checker section advances the theory of interpreted formalisms and real-world types [113] with a formalization based on type theory. This approach makes the expressiveness of higher-order constructive logic available for defining and checking real-world types. Such a checker can be used to establish comprehensive properties. Pluggable type system [114] provide the capability to impose additional type rules on code. Compared with them, our approach exploits the expressive power of dependent types, here with configurations as the "base code" to be further checked. Finding configuration errors has been an active research topic. Mechanisms can be categorized as reactive or proactive. Reactive mechanisms use postmortem analysis of erroneous behaviors and check configuration settings against predefined constraints. Proactive mechanisms try to automatically predict and stop configuration errors early by using techniques such as emulation [115], inference [116], and learning [117–119]. Our pro-active mechanism is unique in exploiting real-world types to exclude configuration errors. Optimizing system performance by configuration search and tuning is not a new idea. Duan et al. [120] proposed to improve database performance by auto-tuning configurations, for example. They sample and profile configurations in a cycle-stealing manner, aborting configuration profiling operations that exceed runtime limits. A type-checker such as ours promises to save significant time in such applications. Configuration search has been used in many domains: energy and delay optimization in embedded

hardware [121]; to reduce cache flushing [19]); robot motion planning [122]; and for connectivity problems [123]. Many approaches account for constraints. Our work is novel in bringing type theory and proof engineering to bear on both expressing and checking constraints.

Learning-based approaches. A large body of previous work estimates system performance by learning performance prediction models. The main challenge of this trend of work is that the training data mainly determines how accurate the trained model will be. Many such approaches suffer from a lack of quality training data. For systems like Hadoop, whose users tend to stick with default settings [2], the obtained traces are likely not to be diverse enough to learn generalized models. Moreover, due to complex parameter interactions, it's difficult to sample a good set of training data. Meinicke et al. [71] studied the configuration spaces of some open-source software systems by running multiple configurations and comparing the differences in control and data flow. They discovered that the interactions were often less than expected but still enough to challenge advanced search strategies. Siegmund et al. [10,14] propose to learn how parameter interactions influence system performance, using different sampling strategies to generate learning sets, and adding interaction terms in learning models for binary and numeric parameters. This approach combines domain knowledge and sampling strategies to generate effective training data. They showed that they can achieve relatively high accuracy compared to previous work. Still, their average error rate ranged from 19% to as high as 37.4%. Others [8, 10, 72] have also tried to detect performance-related interactions to improve model accuracy. Zhang et al. [12] assumes all options are independent and are boolean variables. They then formulate the problem of performance prediction as learning Fourier coefficients of a boolean function. While most of work focus on building accurate performance models, some other work is more final-purpose-driven. Venkataraman et al. [73] aim to predict performance for jobs running on a shared cloud infrastructure. They learn a performance model from previous runs of small instances to predict performance for larger instances. The fundamental assumption in all the work is that to find high-performing configurations one needs a good generalized predictive model. Nair et al. [11]

argued that inaccurate models can also help find better configurations as long as they can preserve the relative ordering of configurations by performance. Our work addresses this problem without trying to learn generalized performance models at all. Our approach addresses *far* larger configuration spaces (estimated to be greater than 10^{25} states) than considered in these earlier efforts.

Metaheuristic search. Our approach traverses the configuration space directly with sampling algorithms to find high-performing configurations. Another body of work leverages meta-heuristic methods to solve complex problems in different domains. Zhang et al. [19] used such algorithms to find better settings for single-level configurable cache to avoid power-consuming cache flushing. They optimized in a space of four parameters including one boolean and three numeric parameters. For robot motion planning, Jaillet et al. [122] used stochastic sampling to find globally low-cost paths for robotics with arbitrary user-given cost functions. Burns et al. [123] used heuristic sampling to solve the connectivity problem in robotic motion planing. Baltz et al. [21] presented their *Optometrist* algorithm, which includes a human in the loop, to find configurations for a plasma fusion reactor. Unique characteristics of our work include use of validated scale-up and scale-out methods to reduce the cost of sampling/search.

Search-based software engineering. In software engineering, many works have utilized heuristic algorithms with promising results. Jia and Harman [59] used automated search to find valuable test cases. McMinn [60] surveyed the application of search techniques for automated test data generation. Weimer et al. [124] reported that they can fix bugs for only $8 each, where they used genetic programming to mutate code statements to produce new code variants as candidate repairs, with test-suite-passing as an objective function. Le [97] used Markov Chain Monte Carlo (MCMC) techniques to generate program variants with different control- and data-flows. They successfully found some bugs in popular compilers like GCC and LLVM. Whittaker and Thomason [125] presented an approach to statistically test software against failures. They used a Markov Chain model to generate test inputs, to study under what usage cases software could fail. Oh et al. [92] worked to find good configurations for software product lines. In this context, the parameters are boolean variables

which correspond to whether a component will be included in a product or not. Vizier [126] is a tool developed at Google for optimizing various systems like machine learning models. It constructs a stack of GP models where each layer refines the model provided by prior layers. This feature allows it runs trails in parallel on distributed system and thus scale well.

Auto-tuning big data framework. A significant amount of work has been done to auto-tune different types of big data frameworks, *e.g.* MapReduce, Spark, *etc.*. Starfish [127] is one of the initial works on Hadoop auto-tuning. It tunes parameters by the predicted results of a cost model, which is built based on average CPU and I/O costs collected by profiling a job with single configuration. As we discussed, accurate performance models are hard to build. If the accuracy of a model is low, it could introduce large errors. Actually, this work [128] has shown that the predicted results of Starfish's cost model could be very different with the actual running time under different reduce tasks settings. Therefore, it could introduce errors that produce bad configurations. Starfish specifically targets the MapReduce framework, because the decision it makes depend on data collected in runtime of mapreduce jobs. Our tool is general and can be easily targeted to work on other systems (as we did for Spark). Babu et al. [63] tune mapreduce parameters by dynamically evaluating a smaller set of sampled data from actual production data that a mapreduce job will process ultimately. It optimizes a job given a cluster and a dataset. Our assumption is much looser, where we do not assume that we are given a dataset. The configurations that we find for a job will work on other datasets. For example, we explore with a smaller dataset A, but validated with another large dataset B, where A has nothing to do with B. Gunther [128] uses vanilla GA plus memoization to identify high-value configurations. But they selected only six important parameters to tune, which greatly reduce the problem size. In contrast, we work on all parameters that related to our objective of performance, which is CPU time in our case. Without knowing how parameters interact with each other to influence results, we cannot exclude any relevant parameter. Jiang et al. [129] conducted a white-box study on Hadoop system to assess how its performance can be affected by different factors. For example, they found that mapreduce jobs often suffer from a lack

of data indexes. Instead, we took a black-box approach to automatically find better values for parameters.

5.7 Conclusions

In this chapter, we proposed to use straightforward heuristic sampling methods like EMCMC in combination with scale-up and scale-out tactics to finding high-performing configurations for complex software systems. We conducted and here report results from a rigorous set of experiments that took about three months of continuous computation on a 5-node Hadoop and Spark clusters. The data that emerged from these experiments provides reasonably strong support for the hypotheses we formulated: configuration space exploration with heuristic search can find high-performing systems. The scale-up and scale-out hypotheses justify the use of small scale jobs to find good configurations for much larger inputs and for other jobs. These results suggest that our approach has significant potential to improve the runtime performance of large and complex software systems in practice.

Chapter 6

Discussion

This dissertation presented three main approaches to improve system performance through design space analysis, performance modeling, and configuration space exploration. We obtained significant improvements in critical engineering domains like ORM and big-data systems. The design and results our evaluations for each of these component efforts are described in the corresponding chapters.

The first set of experimental results make it clear that techniques relying on the single-point strategy, such as conventional ORM tools, produce solutions with strictly and significantly sub-optimal performance relative to actual Pareto fronts, and that our approach is capable of producing strictly and significantly superior designs. Our framework makes the shared architectural and computational structure of a family of similar tools explicit, including its MapReduce computational structure and the filtering of results that are strictly dominated by other results. We captured the generalized structure of this whole tool family using Coq typeclass. Parameters of Coq typeclasses allow variation in the stated dimensions, with propositional laws capturing some of the critical required semantics of the components that one "plugs in" to our framework.

There is a growing need for technologies that can support systematic tradeoff studies to reveal, among others, how system properties in multiple dimensions

vary across implementations, how stakeholders might be impacted, and what implementations might best serve the needs of a given project. We argue this is the holy grail of software design research. Our work takes an important step towards this overarching objective by helping engineers understand important tradeoffs among dynamically measurable properties for important classes of software designs, at meaningful scales within reach of existing synthesis technologies. We envision the ideas set forth in this research to find a broader application in other computing domains as well.

6.1 Limitation

There are limitations in these approaches. Some of them are from the techniques we used, like the model solver in the design space analysis work.

A comment on scalability is in order in our design space analysis work. Relational-logic tradeoff analysis involves exhaustive enumeration of models of bounded relational logic models. This is a *#P-complete* problem: harder than *NP-complete* and equivalent to counting the number of satisfying solutions to an SAT problem. It is intractable in general, and will not scale to complex, integral systems. Yet, model checking tools have clearly demonstrated the potential value in exploring practical uses of solutions to theoretically intractable problems. Relational-logic tradeoff analysis is no panacea. In this work, we observed its potential utility for problems at the scale of individual modules, such as database schemas for ordinary web applications. While the modular architecture of Astronaut supports variation in logic and solvers thus enumeration strategies, we leave the exploration of such variations on our theme to future work. Using Alloy as a constraint solver entails scalability constraints. We can handle object models with tens of classes. Industrial databases often involve thousands of classes. It is unlikely that our current implementation technology will work at that scale. For now, it does have real potential as an aid to smaller-scale system development. That we can present an object model for a realistic web service, synthesize a broad space of ORM strategies, select one based on tradeoff analysis, automatically

obtain a SQL-database setup script, provide it Java EE, and have much of an enterprise-type application up and running with little effort is exciting, even if it does not address (yet) the most demanding needs of industry.

In the performance prediction work, we build prediction models for jobs running on a specific cluster. Such models are not generalized enough to predict the performance of jobs running on different clusters. The reason is that Hadoop logs do not cover features from the hardware layer, but only those from the Hadoop and Hive systems.

In the configuration space exploration work, we had a scalar objective function: CPU time for Hadoop and wall-clock time for Spark. Other applications may have multi-dimensional objectives, with tradeoffs among multiple dimensions of performance. Evolutionary algorithms are known to perform less well in multi-attribute settings. In the general case of system design, one may have dozens of interacting quality attributes that contribute to the system value. Whether techniques such as ours can be adapted to work in or such situations remains a question for further study. Even though, we found configurations that produced massive speed-ups for key classes of MapReduce jobs as represented in an objective benchmark suite. We showed that there is real potential for search using small inputs to produce results that work for much larger problems.

6.2 Threats to Validity

In our performance prediction work, the cluster we collect data from runs different kinds of jobs every day. Some jobs are running periodically. We collected the training data randomly in three months. We expect this method can increase the diversity of the collected data as much as possible. However, it is still possible that the collected data is not diverse enough. It is possible that some jobs run in multiple days during the days that we collected data in, and/or there are some jobs that we never saw. In total, we collected more than thirteen thousands of data samples. There could be some features do not appear enough times to be captured by our algorithm. We could, in theory, be in the situation of “curse

of dimensionality" [130]. The method we used to approximate the complexity of Hive queries is relatively simple. We choose about twenty kinds of tokens extracted from a Hive query based on our understanding. It's possible that we missed some tokens which are also important. A systematic study of the logic complexity of Hive queries can probably produce more accurate complexity, which will be more helpful in our project. How to estimate the job complexity more accurately without much cost would be good future work.

Due to the nature of dynamic evaluation, it is possible that the experimental results are affected by uncontrolled factors on hardware platforms. In our experiments, we adopted some strategies to mitigate such unseen factors. For example, we make sure that no other programs are running on the experimental platform while we are running experiments. We also run each dynamic evaluation three times to get average performance as a final result. In this work, we choose a subset of all parameters to study with domain knowledge. It is possible that we missed some important ones. To mitigate this threat, we referred to many previous works [18] on Hadoop, and have included all parameters studied by other researchers in our parameter set.

6.3 Future Work

Our design space analysis framework, Astronaut, is designed as a generalized suite that can be easily applied in other domains. This could be done by plugging in different components into the framework. In the empirical study section, we plugged in two different DBMSs, namely MySQL and PostgreSQL, to initially show the generalization of Astronaut. As part of the future work, one could develop different specifications and/or model solvers to apply it in other domains.

The sensitivity analysis results in our configuration space exploration strongly suggest that there are non-negligible interactions among parameter values in determining the outputs of our objective functions. In future work, we hope to learn how to infer modularity properties of such objective functions, to

enable parallel optimization of coupled parameter clusters, so as to significantly reduce the sizes of the state spaces to be sampled. For example, we can try to group parameters by which computation phase they are involved with, e.g., parameters that only involve the *map* step might form a mostly independent cluster (module).

Chapter 7

Conclusion

This dissertation developed and evaluated a set of propositions about how to improve the performance of large-scale and complex big data software systems using design and configuration space exploration techniques and statistical learning-based prediction models where prediction is based to a considerable extent on configuration.

In Chapter 3, we talked about how to synthesize and evaluate the whole design spaces exhaustively. Starting from a given system specification, we synthesize all designs and common test suites which will be used to assess the designs later. Since a design space is typically large, meaning there are a vast number of designs, we developed an automated general framework for conducting design space analysis. We applied this technique in the ORM domain for finding Pareto-optimal database schemas. The evaluation results on seven subject systems and two common DBMSs show that our technology is promising in finding Pareto-optimal system designs in both time and space performance for modest-scale problems. The key to scaling up is to determine how to decompose complex systems in modest-scale subsystems that can be handled by our methods (when possible).

Chapter 4 presents a novel approach to improve the accuracy of performance prediction models using the semantic meanings of configuration parameters to

clean the training data. We applied this approach in predicting CPU time and physical memory consumption for MapReduce jobs. We built models using different data sets that are cleaned with and without our approach. The results show that our approach can significantly improve model accuracy.

In Chapter 5, we proposed to use meta-heuristic algorithms to search the configuration space of a software system to find configurations to improve its performance. This approach samples configurations and guides the search procedure with the evaluated performance information. We evaluated three different sampling strategies, namely random sampling as the baseline, genetic algorithm, and EMCMC. We applied this approach in Hadoop and found that EMCMC performs best out of these three algorithms. It can find configurations that significantly improve the time performance of regular MapReduce jobs. To save the search cost, our approach explores a configuration space with a small size of input data and verify found candidates with larger datasets. We also found that similar jobs can obtain performance gain from the same configuration.

Overall, then, the results of experimental evaluation presented in previous chapters tend to support the thesis of this dissertation. Our approaches to performance predication and optimization of big data systems focused on the role and impact of configuration appears to have real potential to improve real-world performance in critical and costly big data infrastructure systems.

Bibliography

[1] Apache hadoop. `http://hadoop.apache.org/`, 2017. Accessed: 2016-08-06.

[2] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. *Proceedings of the VLDB Endowment*, 6(10):853–864, 2013.

[3] D.-I. Kang, R. Gerber, L. Golubchik, J. K. Hollingsworth, and M. Saksena. A software synthesis tool for distributed embedded system design. *SIGPLAN Not.*, 34(7), May 1999.

[4] Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, New York, NY, USA. ACM.

[5] I. Assayad, V. Bertin, F x. Defaut, Ph. Gerner, O. Quévreux, and S. Yovine. Jahuel: A formal framework for software synthesis. In *in ICFEM'05*, 2005.

[6] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 313–326, New York, NY, USA, January 2010. ACM.

[7] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6), Aug 2012.

[8] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 301–311. IEEE, 2013.

[9] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 342–352. IEEE, 2015.

[10] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Software Engi-*

neering (ICSE), 2012 34th International Conference on, ICSE '12, pages 167–177, Piscataway, NJ, USA, 2012. IEEE, IEEE Press.

[11] V. Nair, T. Menzies, N. Siegmund, and S. Apel. Using bad learners to find good configurations. *ArXiv e-prints*, February 2017.

[12] 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, 2015. IEEE, IEEE Computer Society.

[13] Ge Song, Zide Meng, Fabrice Huet, Frederic Magoules, Lei Yu, and Xuelian Lin. A hadoop mapreduce performance prediction method. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 820–825. IEEE, 2013.

[14] 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*, pages 284–294. ACM, 2015.

[15] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[16] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *16th International Conference on VLSI Design, 2003. Proceedings.* Institute of Electrical & Electronics Engineers (IEEE), 2003.

[17] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 105–114. ACM, 2010.

[18] Ailton S Bonifacio, Andre Menolli, and Fabiano Silva. Hadoop mapreduce configuration parameters and system performance: a systematic review. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2014.

[19] Chuanjun Zhang, Frank Vahid, and Roman Lysecky. A self-tuning cache architecture for embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(2):407–425, 2004.

[20] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[21] EA Baltz, E Trask, M Binderbauer, M Dikovsky, H Gota, R Mendoza, JC Platt, and PF Riley. Achievement of sustained net plasma heating in a fusion experiment with the optometrist algorithm. *Scientific Reports*, 7, 2017.

[22] Tripti Saxena and Gabor Karsai. Mde-based approach for generalizing design space exploration. In *International Conference on Model Driven Engineering Languages and Systems*, pages 46–60. Springer, 2010.

[23] Hamid Bagheri and Kevin Sullivan. Spacemaker: Practical formal synthesis of tradeoff spaces for object-relational mapping. In *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering*, San Francisco Bay, USA, 2012.

[24] Aline Lúcia Baroni, Coral Calero, Mario Piattini, and O Brito E Abreu. A formal definition for ObjectRelational database metrics. In *Proceedings of the 7th International Conference on Enterprise Information System*, 2005.

[25] Stefan Holder, Jim Buchan, and Stephen G. MacDonell. Towards a metrics suite for Object-Relational mappings. *Model-Based Software and Data Integration*, CCIC 8:43–54, 2008.

[26] Cristian Cadar, Peter Pietzuch, and Alexander L. Wolf. Multiplicity computing: a vision of software engineering for next-generation computing platform applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 81–86, New York, NY, USA, 2010. ACM.

[27] Luca Cabibbo and Antonio Carosi. Managing inheritance hierarchies in Object/Relational mapping tools. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05)*, pages 135–150, 2005.

[28] Wolfgang Keller. Mapping objects to tables - a pattern language. In *Proc. of the European Pattern Languages of Programming Conference*, 1997.

[29] Stephan Philippi. Model driven generation and testing of object-relational mappings. *Journal of Systems and Software*, 77(2):193–207, 2005.

[30] Sean Quan Lau. *Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates*. Master's thesis, University of Waterloo, Canada, 2006.

[31] Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, platforms and possibilities: Towards generic automation for MDA. In *Proceedings of International Conference on Embedded Software*, 2010.

[32] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 238–247. IEEE Computer Society, 2008.

[33] S. E. Stemler. A comparison of consensus, consistency, and measurement approaches to estimating interrater reliability. *Practical Assessment, Research and Evaluation*, 9(4), 2004.

[34] Hamid Bagheri, Chong Tang, and Kevin Sullivan. Trademaker: Automated dynamic analysis of synthesized tradespaces. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 106–116, New York, NY, USA, 2014. ACM, ACM.

[35] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[36] Bas Spitters and Eelis Van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(04), 2011.

[37] Álvaro Pelayo, Michael A Warren, and To Vladimir Voevodsky. Homotopy type theory. *Gazette des Mathematiciens*, (142), 2014.

[38] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the coq proof assistant*. MIT Press, 2013.

[39] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[40] Flagship docs 4. Accessed: 2015-10-22.

[41] Erik Andersen, Sumit Gulwani, and Zoran Popovic. A trace-based framework for analyzing and synthesizing educational progressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, New York, NY, USA. ACM.

[42] Shriram Krishnamurthi, Kathi Fisler, Daniel J. Dougherty, and Daniel Yoo. Alchemy: transmuting base alloy specifications into implementations. In *Proceedings of FSE'08*, pages 158–169, 2008.

[43] Alcino Cunha and Hugo Pacheco. Mapping between alloy specifications and database implementations. In *Proceedings of the Seventh International Conference on Software Engineering and Formal Methods (SEFM'09)*, pages 285–294, 2009.

[44] Christopher Ireland, David Bowers, Mike Newton, and Kevin Waugh. Understanding object-relational mapping: A framework based approach. *International Journal on Advances in software*, 2:202–216, 2009.

[45] Mauro Luigi Drago, Carlo Ghezzi, and Raffaela Mirandola. A quality driven extension to the QVT-relations transformation language. *Computer Science - Research and Development*, 2011.

[46] Florian Heidenreich, Christian Wende, and Birgit Demuth. A framework for generating query language code from OCL invariants. *Electronic Communications of the EASST*, 9, November 2007.

[47] Mohammad Badawy and Karel Richta. Deriving triggers from UML/OCL specification. In *Information Systems Development*, pages 305–315. Springer US, 2002.

[48] Harith T. Al-Jumaily, Dolores Cuadra, and Paloma Martínez. Software architecture: Ocl2trigger: Deriving active mechanisms for relational databases using model-driven architecture. *J. Syst. Softw.*, 81(12), December 2008.

[49] Birgit Demuth, Heinrich Hussmann, and Sten Loecher. OCL as a specification language for business rules in database applications. In *Proceedings of the UML 2001–The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 104–117, 2001.

[50] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 341—352, New York, NY, USA, 2007. ACM.

[51] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for DBMS testing. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1721–1725, 2006.

[52] Claudio de la Riva, María José Suárez-Cabal, and Javier Tuya. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 67–74, Cape Town, South Africa, 2010. ACM.

[53] TPC benchmarks. http://www.tpc.org.

[54] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In *Genetic Programming*. Springer Science + Business Media, 2014.

[55] Egor Bondarev, Michel RV Chaudron, and Erwin A de Kock. Exploring performance trade-offs of a jpeg decoder using the deepcompass framework. In *Proceedings of the 6th international workshop on Software and performance*, pages 153–163. ACM, 2007.

[56] Aldeida Aleti, Stefan Bjornander, Lars Grunske, and Indika Meedeniya. Archeopterix: An extendable tool for architecture optimization of aadl models. In *Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES'09. ICSE Workshop on*, pages 61–71. IEEE, February 2009.

[57] Anne Martens, Heiko Koziolek, Steffen Becker, and Ralf Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *Proceedings of the 1st Int. Conf. on Performance Engineering*, pages 105–116. ACM, February 2010.

[58] Mark Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, FOSE '07, Washington, DC, USA, 2007. IEEE Computer Society.

[59] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10), 2009.

[60] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2), 2004.

[61] Jim Manzi. *Uncontrolled: The surprising payoff of trial-and-error for business, politics, and society*. Basic Books (AZ), 2012.

[62] Mike Moran. *Do it wrong quickly: how the web changes the old marketing rules*. IBM Press, 2007.

[63] Shivnath Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 137–142. ACM, 2010.

[64] Zhenhua Guo, Geoffrey Fox, Mo Zhou, and Yang Ruan. Improving resource utilization in mapreduce. In *2012 IEEE International Conference on Cluster Computing*, pages 402–410. IEEE, 2012.

[65] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[66] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H. Byers. Big data: The next frontier for innovation, competition, and productivity, May 2011.

[67] Rajkumar Buyya et al. High performance cluster computing: Architectures and systems (volume 1). *Prentice Hall, Upper SaddleRiver, NJ, USA*, 1:999, 1999.

[68] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[69] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[70] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[71] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: measuring interactions in highly-configurable systems. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 483–494. IEEE, 2016.

[72] Sven Apel, Alexander von Rhein, Thomas Thüm, and Christian Kästner. Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399–2409, 8 2013.

[73] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[74] Enrico Barbierato, Marco Gribaudo, and Mauro Iacono. Modeling apache hive based applications in big data architectures. In *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '13, pages 30–38, ICST, Brussels, Belgium, Belgium, 2013. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[75] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Performance modeling of mapreduce jobs in heterogeneous cloud environments. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 839–846. IEEE, 2013.

[76] Mukhtaj Khan, Yong Jin, Maozhen Li, Yang Xiang, and Changjun Jiang. Hadoop performance modeling for job estimation and resource provisioning. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):441–454, 2016.

[77] Mapr control system. `https://www.mapr.com/products/product-overview/mapr-control-system`. Accessed: 2016-08-15.

[78] Beautiful soup. `https://www.crummy.com/software/BeautifulSoup/`. Accessed: 2016-08-15.

[79] Malik Yousef, Müşerref Duygu Saçar Demirci, Waleed Khalifa, and Jens Allmer. Feature selection has a large impact on one-class classification accuracy for microrna s in plants. *Advances in bioinformatics*, 2016, 2016.

[80] Andreas Janecek, Wilfried Gansterer, Michael Demel, and Gerhard Ecker. On the relationship between feature selection and classification accuracy. In *New Challenges for Feature Selection in Data Mining and Knowledge Discovery*, pages 90–105, 2008.

[81] Esra Mahsereci Karabulut, Selma Ayşe Özel, and Turgay Ibrikci. A comparative study on the effect of feature selection on classification accuracy. *Procedia Technology*, 1:323–327, 2012.

[82] Hive parser api. `https://hive.apache.org/javadocs/r0.10.0/api/org/apache/hadoop/hive/ql/parse/HiveParser.html`. Accessed: 2016-08-15.

[83] Numpy. `http://www.numpy.org/`. Accessed: 2016-08-15.

[84] scikit-learn. `http://scikit-learn.org//`. Accessed: 2016-08-15.

[85] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 307–319. ACM, 2015.

[86] Zhen Jia, Chao Xue, Guancheng Chen, Jianfeng Zhan, Lixin Zhang, Yonghua Lin, and Peter Hofstee. Auto-tuning spark big data workloads on power8: Prediction-based dynamic smt threading. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 387–400. IEEE, 2016.

[87] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[88] Jordan J Louviere, David Pihlens, and Richard Carson. Design of discrete choice experiments: a discussion of issues that matter in future applied research. *Journal of Choice Modelling*, 4(1):1–8, 2011.

[89] Shrinivas B. Joshi. Apache hadoop performance-tuning methodologies and best practices. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 241–242, New York, NY, USA, 2012. ACM.

[90] Gary M Weiss and Ye Tian. Maximizing classifier utility when there are data acquisition and modeling costs. *Data Mining and Knowledge Discovery*, 17(2):253–282, 2008.

[91] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[92] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 61–71. ACM, 2017.

[93] Adrian A Canutescu and Roland L Dunbrack. Cyclic coordinate descent: A robotics algorithm for protein loop closure. *Protein science*, 12(5):963–972, 2003.

[94] Po-Ling Loh and Martin J Wainwright. Regularized m-estimators with nonconvexity: Statistical and algorithmic theory for local optima. In *Advances in Neural Information Processing Systems*, pages 476–484, 2013.

[95] Mădălina M Drugan and Dirk Thierens. Evolutionary markov chain monte carlo. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 63–76. Springer, 2003.

[96] W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[97] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *ACM SIGPLAN Notices*, volume 50, pages 386–399. ACM, 2015.

[98] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. *Machine learning*, 50(1-2):5–43, 2003.

[99] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[100] Reza Akbari and Koorush Ziarati. A multilevel evolutionary algorithm for optimizing numerical functions. *International Journal of Industrial Engineering Computations*, 2(2):419–430, 2011.

[101] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[102] Chong Tang, Kevin Sullivan, Jian Xiang, Trent Weiss, and Baishakhi Ray. Interpreted formalisms for configurations. *arXiv preprint arXiv:1712.04982*, 2017.

[103] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[104] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. Scalable product line configuration: A straw to break the camel's back. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 465–474. IEEE, 2013.

[105] Jian Xiang, John Knight, and Kevin Sullivan. Synthesis of logic interpretations. In *High Assurance Systems Engineering (HASE), 2016 IEEE 17th International Symposium on*, pages 114–121. IEEE, 2016.

[106] Jian Xiang, John Knight, and Kevin Sullivan. Is my software consistent with the real world? In *High Assurance Systems Engineering (HASE), 2017 IEEE 18th International Symposium on*, pages 1–4. IEEE, 2017.

[107] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[108] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.

[109] Nikhil Khadke, Michael P Kasick, Soila Kavulya, Jiaqi Tan, and Priya Narasimhan. Transparent system call based performance debugging for cloud computing. In *MAD*, 2012.

[110] Eunjung Yoon and Anna Squicciarini. Toward detecting compromised mapreduce workers through log analysis. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 41–50. IEEE, 2014.

[111] Michael P Kasick, Keith A Bare, Eugene E Marinelli III, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. System-call based problem diagnosis for pvfs. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF ELECTRICAL AND COMPUTER ENGINEERING, 2009.

[112] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering*, pages 1–31, 2017.

[113] Jian Xiang. *Interpreted Formalism: Towards System Assurance and the Real-World Semantics of Software*. PhD thesis, University of Virginia, 2016.

[114] Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690. ACM, 2011.

[115] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *OSDI*, pages 619–634, 2016.

[116] Xiangyang Xu, Shanshan Li, Yong Guo, Wei Dong, Wang Li, and Xiangke Liao. Automatic type inference for proactive misconfiguration prevention. In *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering*, 2017.

[117] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. Probabilistic automated language learning for configuration files. In *International Conference on Computer Aided Verification*, pages 80–87. Springer, 2016.

[118] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, pages 28–28. USENIX Association, 2011.

[119] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. *ACM SIGPLAN Notices*, 49(4):687–700, 2014.

[120] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[121] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Multi-objective design space exploration of embedded systems. *Journal of Embedded Computing*, 1(3):305–316, 2005.

[122] Léonard Jaillet, Juan Cortés, and Thierry Siméon. Sampling-based path planning on configuration-space costmaps. *IEEE Transactions on Robotics*, 26(4):635–646, 2010.

[123] Brendan Burns and Oliver Brock. Toward optimal configuration space sampling. In *Robotics: Science and Systems*, pages 105–112. Citeseer, 2005.

[124] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[125] James A Whittaker and Michael G Thomason. A markov chain model for statistical software testing. *IEEE Transactions on Software engineering*, 20(10):812–824, 1994.

[126] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495. ACM, 2017.

[127] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[128] Guangdeng Liao, Kushal Datta, and Theodore L Willke. Gunther: Search-based auto-tuning of mapreduce. In *European Conference on Parallel Processing*, pages 406–419. Springer, 2013.

[129] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: an in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.

[130] David L Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS Math Challenges Lecture*, pages 1–32, 2000.

[131] Ruby on rails. `http://rubyonrails.org`. Accessed: 2015-10-22.

[132] Django. `https://www.djangoproject.com`. Accessed: 2015-10-22.

[133] Cheng Huang, Minghua Chen, and Jin Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *ACM Transactions on Storage (TOS)*, 9(1):3, 2013.

[134] Adam Michael Ross. Managing unarticulated value: changeability in multi-attribute tradespace exploration. *Engineering Systems Division*, 361, 2006.

[135] Alan Davis, Scott Overmyer, Kathleen Jordan, Joseph Caruso, Fatma Dandashi, Anhtuan Dinh, Gary Kincaid, Glen Ledeboer, Patricia Reynolds, Pradip Sitaram, et al. Identifying and measuring quality in a software requirements specification. In *Software Metrics Symposium, 1993. Proceedings., First International*, pages 141–152. IEEE, 1993.

[136] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[137] Adam M Ross, Daniel E Hastings, Joyce M Warmkessel, and Nathan P Diller. Multi-attribute tradespace exploration as front end for effective space system design. *Journal of Spacecraft and Rockets*, 41(1):20–28, 2004.

[138] Murray Woodside, Greg Franks, and Dorina C Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, pages 171–187. IEEE Computer Society, 2007.

[139] Charles J Geyer. Practical markov chain monte carlo. *Statistical science*, pages 473–483, 1992.

[140] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[141] Zong Woo Geem, Joong Hoon Kim, and GV Loganathan. A new heuristic optimization algorithm: harmony search. *Simulation*, 76(2):60–68, 2001.

[142] Melanie Mitchell, John H Holland, and Stephanie Forrest. When will a genetic algorithm outperform hill climbing? *Ann Arbor*, 1001:48109, 1993.

[143] Abhinav Kamra, Vishal Misra, and Erich M Nahum. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on*, pages 47–56. IEEE, 2004.

[144] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.

[145] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*, pages 1–10. ACM, 2010.

[146] Cemal Yilmaz, Myra B Cohen, and Adam A Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 32(1):20–34, 2006.

[147] Matteo Dell'Amico, Damiano Carra, Mario Pastorelli, and Pietro Michiardi. Revisiting size-based scheduling with estimated job sizes. In *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 411–420. IEEE, 2014.

[148] Archana Ganapathi, Yanpei Chen, Armando Fox, Randy Katz, and David Patterson. Statistics-driven workload modeling for the cloud. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 87–92. IEEE, 2010.

[149] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*, pages 592–603. IEEE, 2009.

[150] Changlong Li, Hang Zhuang, Kun Lu, Mingming Sun, Jinhong Zhou, Dong Dai, and Xuehai Zhou. An adaptive auto-configuration tool for hadoop. In *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*, pages 69–72. IEEE, 2014.

[151] Ivanilton Polato, Reginaldo Ré, Alfredo Goldman, and Fabio Kon. A comprehensive view of hadoop research—a systematic literature review. *Journal of Network and Computer Applications*, 46:1–25, 2014.

[152] Seyed Reza Pakize. A comprehensive view of hadoop mapreduce scheduling algorithms. *International Journal of Computer Networks & Communications Security*, 2(9):308–317, 2014.

[153] Ibrahim Abaker Targio Hashem, Nor Badrul Anuar, Abdullah Gani, Ibrar Yaqoob, Feng Xia, and Samee Ullah Khan. Mapreduce: Review and open challenges. *Scientometrics*, pages 1–34, 2016.

[154] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244. ACM, 2011.

[155] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Parameterizable benchmarking framework for designing a mapreduce performance model. *Concurrency and Computation: Practice and Experience*, 26(12):2005–2026, 2014.

[156] Nikzad Babaii Rizvandi, Javid Taheri, Reza Moraveji, and Albert Y Zomaya. On modelling and prediction of total cpu usage for applications in mapreduce environments. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 414–427. Springer, 2012.

[157] Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *MASCOTS*, volume 9, pages 1–11, 2009.

[158] Mingyuan An, Yang Wang, and Weiping Wang. Using index in the mapreduce framework. In *Web Conference (APWEB), 2010 12th International Asia-Pacific*, pages 52–58. IEEE, 2010.

[159] Suhel Hammoud, Maozhen Li, Yang Liu, Nasullah Khalid Alham, and Zelong Liu. Mrsim: A discrete event based mapreduce simulator. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh International Conference on*, volume 6, pages 2993–2997. IEEE, 2010.

[160] Hailong Yang, Zhongzhi Luan, Wenjun Li, Depei Qian, and Gang Guan. Statistics-based workload modeling for mapreduce. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2043–2051. IEEE, 2012.

[161] Nikzad Babaii Rizvandi, Javid Taheri, Reza Moraveji, and Albert Y Zomaya. Network load analysis and provisioning of mapreduce applications. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 161–166. IEEE, 2012.

[162] Di Xie, Y Charlie Hu, and Ramana Rao Kompella. On the performance projectability of mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 301–308. IEEE, 2012.

[163] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 18. ACM, 2011.

[164] Hailong Yang, Zhongzhi Luan, Wenjun Li, and Depei Qian. Mapreduce workload modeling with statistical approach. *Journal of grid computing*, 10(2):279–310, 2012.

[165] Jinquan Dai, Jie Huang, Shengsheng Huang, Bo Huang, and Yan Liu. Hitune: dataflow-based performance analysis for big data cloud. *Proc. of the 2011 USENIX ATC*, pages 87–100, 2011.

[166] Yanpei Chen, Archana Sulochana Ganapathi, Rean Griffith, and Randy H Katz. Towards understanding cloud performance tradeoffs using statistical workload analysis and replay. *University of California at Berkeley, Technical Report No. UCB/EECS-2010-81*, 2010.

[167] Palden Lama and Xiaobo Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th international conference on Autonomic computing*, pages 63–72. ACM, 2012.

[168] Yingjie Shi, Xiaofeng Meng, Jing Zhao, Xiangmei Hu, Bingbing Liu, and Haiping Wang. Benchmarking cloud-based data management systems. In *Proceedings of the second international workshop on Cloud data management*, pages 47–54. ACM, 2010.

[169] Guanying Wang, Ali R Butt, Prashant Pandey, and Karan Gupta. Using realistic simulation for performance analysis of mapreduce setups. In *Proceedings of the 1st ACM workshop on Large-Scale system and application performance*, pages 19–26. ACM, 2009.

[170] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Benchmarking approach for designing a mapreduce performance model. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 253–258. ACM, 2013.

[171] Herodotos Herodotou, Fei Dong, and Shivnath Babu. Mapreduce programming and cost-based optimization? crossing this chasm with starfish. *Proceedings of the VLDB Endowment*, 4(12):1446–1449, 2011.

[172] Hadoop fair scheduler. `https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html`. Accessed: 2016-08-20.

[173] Hadoop capacity scheduler. `https://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html`. Accessed: 2016-08-20.

[174] Pandas. `http://pandas.pydata.org/`. Accessed: 2016-08-15.

[175] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103, Washington, DC, USA, 2010. IEEE Computer Society.

[176] Shouvik Bardhan and D Menasce. Queuing network models to predict the completion time of the map phase of mapreduce jobs. In *Proceedings of the Computer Measurement Group International Conference*. Citeseer, 2012.

[177] Gunho Lee, Byung-Gon Chun, and H. Katz. Heterogeneity-aware resource allocation and scheduling in the cloud. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 4–4, Berkeley, CA, USA, 2011. USENIX Association.

[178] Nam P Suh. *The principles of design*. Number 6. Oxford University Press on Demand, 1990.

[179] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[180] Carliss Young Baldwin and Kim B Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[181] Kevin J Sullivan, William G Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 99–108. ACM, 2001.

[182] Steven D Eppinger and Tyson R Browning. *Design structure matrix methods and applications*. MIT press, 2012.

[183] Tyson R Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering management*, 48(3):292–306, 2001.

[184] Terasort. http://sortbenchmark.org/. (Accessed on 08/19/2017).

[185] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[186] Tony Givargis and Frank Vahid. Platune: a tuning framework for system-on-a-chip platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(11):1317–1327, 2002.

[187] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[188] Hamid Bagheri, Chong Tang, and Kevin Sullivan. Automated synthesis and dynamic analysis of tradeoff spaces for object-relational mapping. *IEEE Transactions on Software Engineering*, 43(2):145–163, 2017.

[189] Siddhartha Chib and Edward Greenberg. Understanding the metropolis-hastings algorithm. *The american statistician*, 49(4):327–335, 1995.

[190] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by

fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[191] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.

[192] Faming Liang and Wing Hung Wong. Evolutionary monte carlo: Applications to c p model sampling and change point problem. *Statistica sinica*, pages 317–342, 2000.

[193] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 131–140. ACM, 2011.

[194] Jian Xiang, John Knight, and Kevin Sullivan. Real-world types and their application. In *International Conference on Computer Safety, Reliability, and Security*, pages 471–484. Springer, 2014.

[195] Carl A Gunter, Elsa L Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.

[196] Mike Nahas. A tutorial by mike nahas. https://coq.inria.fr/tutorial-nahas. Accessed on 08/19/2018.

[197] Ariel Shemaiah Rabkin. *Using program analysis to reduce misconfiguration in open source systems software*. University of California, Berkeley, 2012.

[198] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM SIGPLAN Notices*, volume 41, pages 42–54. ACM, 2006.

[199] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *ACM SIGARCH Computer Architecture News*, volume 42 of *ASPLOS '14*, pages 639–652, New York, NY, USA, 2014. ACM, ACM.