Algorithm Selection, Scheduling and Configuration of Boolean Constraint Solvers

---

Dissertation

von

T. Marius Lindauer (M.Sc.)

Universität Potsdam
Institut für Informatik
Professur Wissensverarbeitung und Informationssysteme

Aufgabenstellung und Betreuung:
Prof. Dr. Torsten Schaub, University of Potsdam
Prof. Dr. Holger H. Hoos, University of British Columbia
Disputation am 08. Januar 2015

Eingereicht zur Erlangung des akademischen Grades Dr. rer. nat in der Wissenschaftsdisziplin "Wissensverarbeitung und Informationssysteme"

Potsdam, den 23. Februar 2015

Lindauer, T. Marius
manju@cs.uni-potsdam.de
Algorithm Selection, Scheduling and Configuration of Boolean Constraint Solvers
Dissertation, Institut für Informatik
Universität Potsdam, 2015

Published online at the
Institutional Repository of the University of Potsdam:
http://publishup.uni-potsdam.de/opus4-ubp/frontdoor/index/index/docId/7126
URN urn:nbn:de:kobv:517-opus4-71260
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-71260

Acknowledgement

First of all, I would like to thank my two advisors, Torsten Schaub and Holger Hoos. I have learned a lot from both in these four years, particularly they did not always have the same opinion. Furthermore, a big thanks to Klemens Kittan who supported me on all issues and questions regarding our cluster. Without his support, most of my experiments would not have been possible. Sabine Hübner, our secretary, always supported me with administration issues, such as business travel.

A further thanks to all my proof readers, Frank Hutter, Thomas Jung, Simon Kiertscher, Ina Lindauer and Max Möller. And of course, I also enjoyed my time in Potsdam with my colleagues, Benjamin Andres, Steffen Christgau, Martin Gebser, Holger Jost, Roland Kaminski, Benjamin Kaufmann, Simon Kiertscher, Arne König, Philipp Obermeier, Max Ostrowski, Javier Davila and Orkunt Sabuncu.

Last but not least, I give a huge thanks to my daughter, who always had a big grin for me, my wife, who relieved me whenever it was necessary, my mother, who always gave me great advice for all questions of daily life and my stepfather, who was always a model for me.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt, sie nicht anderweitig zu Prüfungszwecken vorgelegt wurde, keine anderen als die angegebenen Hilfsmittel verwendet habe und alle bereits publizierten Kapitel explizit als solche gekennzeichnet sind. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, den 23. Februar 2015

T. Marius Lindauer

Abstract

Boolean constraint solving technology has made tremendous progress over the last decade, leading to industrial-strength solvers, for example, in the areas of answer set programming (ASP), the constraint satisfaction problem (CSP), propositional satisfiability (SAT) and satisfiability of quantified Boolean formulas (QBF). However, in all these areas, there exist multiple solving strategies that work well on different applications; no strategy dominates all other strategies. Therefore, no individual solver shows robust state-of-the-art performance in all kinds of applications. Additionally, the question arises how to choose a well-performing solving strategy for a given application; this is a challenging question even for solver and domain experts. One way to address this issue is the use of portfolio solvers, that is, a set of different solvers or solver configurations. We present three new automatic portfolio methods: (i) automatic construction of parallel portfolio solvers (ACPP) via algorithm configuration, (ii) solving the NP-hard problem of finding effective algorithm schedules with Answer Set Programming (*aspeed*), and (iii) a flexible algorithm selection framework (*claspfolio* 2) allowing for fair comparison of different selection approaches. All three methods show improved performance and robustness in comparison to individual solvers on heterogeneous instance sets from many different applications. Since parallel solvers are important to effectively solve hard problems on parallel computation systems (e.g., multi-core processors), we extend all three approaches to be effectively applicable in parallel settings. We conducted extensive experimental studies different instance sets from ASP, CSP, MAXSAT, Operation Research (OR), SAT and QBF that indicate an improvement in the state-of-the-art solving heterogeneous instance sets. Last but not least, from our experimental studies, we deduce practical advice regarding the question when to apply which of our methods.

Zusammenfassung

Bool'sche Solver Technologie machte enormen Fortschritt im letzten Jahrzehnt, was beispielsweise zu industrie-relevanten Solvern auf der Basis von Antwortmengenprogrammierung (ASP), dem Constraint Satisfcation Problem (CSP), dem Erfüllbarkeitsproblem für aussagenlogische Formeln (SAT) und dem Erfüllbarkeitsproblem für quantifizierte boolesche Formeln (QBF) führte. Allerdings gibt es in all diesen Bereichen verschiedene Lösungsstrategien, welche bei verschiedenen Anwendungen unterschiedlich effizient sind. Dabei gibt es keine einzelne Strategie, die alle anderen Strategien dominiert. Das führt dazu, dass es keinen robusten Solver für das Lösen von allen möglichen Anwendungsprobleme gibt. Die Wahl der richtigen Strategie für eine neue Anwendung ist eine herausforderne Problemstellung selbst für Solver- und Anwendungsexperten. Eine Möglichkeit, um Solver robuster zu machen, sind Portfolio-Ansätze. Wir stellen drei automatisch einsetzbare Portfolio-Ansätze vor: (i) automatische Konstruktion von parallelen Portfolio-Solvern (ACPP) mit Algorithmen-Konfiguration, (ii) das Lösen des *NP*-harten Problems zur Algorithmen-Ablaufplanung (*aspeed*) mit ASP, und (iii) ein flexibles Algorithmen-Selektionsframework (*claspfolio* 2), was viele Techniken von Algorithmen-Selektion parametrisiert implementiert und eine faire Vergleichbarkeit zwischen Ihnen erlaubt. Alle drei Methoden verbessern die Robustheit des Solvingprozesses für heterogenen Instanzmengen bestehend aus unterschiedlichsten Anwendungsproblemen. Parallele Solver sind zunehmend der Schlüssel zum effektiven Lösen auf multi-core Maschinen. Daher haben wir all unsere Ansätze auch für den Einsatz auf parallelen Architekturen erweitert. Umfangreiche Experimente auf ASP, CSP, MAXSAT, Operation Research (OR), SAT und QBF zeigen, dass der Stand der Technik durch verbesserte Performanz auf heterogenen Instanzmengen verbessert wurde. Auf Grundlage dieser Experimente leiten wir auch Ratschläge ab, in welchen Anwendungsszenarien welches unserer Verfahren angewendet werden sollte.

Contents

1 Introduction

Boolean constraint solving technology has made tremendous progress over the last decade, leading to industrial-strength solvers. Although this advance in technology has occurred to a large extent in the area of propositional satisfiability (SAT) (Biere, Heule, van Maaren, & Walsh, 2009), it also led to significant boosts in neighboring areas, like Answer Set Programming (ASP) (Baral, 2003), Pseudo-Boolean Solving (Biere et al., 2009, Chapter 22), and even (multi-valued) Constraint Solving (Tamura, Taga, Kitagawa, & Banbara, 2009). However, in all these areas multiple solving strategies exist that are complementary to each other, that is, no strategy dominates all other strategies on all kind of problems. This holds in many sub-communities of artificial intelligence and is well supported by empirical results in the literature, for example, propositional satisfiability (SAT) (Xu, Hutter, Hoos, & Leyton-Brown, 2012a), constraint satisfaction (CSP) (O'Mahony, Hebrard, Holland, Nugent, & O'Sullivan, 2008), AI planning (Helmert, Röger, & Karpas, 2011), and supervised machine learning (Thornton, Hutter, Hoos, & Leyton-Brown, 2013).

Boolean constraint solvers are of special interest in this context, because they are implemented to efficiently solve NP-hard (or even harder) problems such that runtime optimization is crucial for them. However, the performance of a solving strategy differs on different problems. Therefore, applying a badly chosen solving strategy to a given problem can result in a drastically longer solving time.[1] Even in the average case, using the right solving strategy can improve the runtime of a Boolean constraint solver by orders of magnitude (see, for example, Hutter, Hoos, and Leyton-Brown (2010)).

Our running example to illustrate this issue is Answer Set Programming (ASP; Baral (2003), Gebser, Kaminski, Kaufmann, and Schaub (2012)). ASP is a form of declarative programming with roots in knowledge representation, non-monotonic reasoning and constraint solving. In contrast to many other constraint solving domains (for example, the satisfiability problem), ASP provides a rich, yet simple declarative modeling language in which problems in NP, problems in NP^{NP} and hierarchical optimization problems can be expressed. Since ASP provides a declarative language, a user does not have to specify how to solve a given problem, but he merely hast to provide a representation of the problem (without specifying how to solve it). However, for practical usage, the average solving performance is often crucial, and average performance strongly depends on the right choice of the solving strategy (besides modelling techniques). Since only very few experts have enough expertise to find a well-performing solving strategy, users without expert knowledge are still confronted with the challenge to decide how to solve his problem in form of selecting a solving strategy. Hence, the declarativeness of ASP is still limited. The motivation of this dissertation is to provide mechanisms to overcome this limitation and provide tools to automatically find well-performing solving strategies for a given

[1] For example, stochastic local search solvers can solve efficiently randomly generated SAT instances that cannot be solved by CDCL-based solvers within orders of magnitude more time; see, for example, the results of the international SAT Competition (`http://www.satcompetition.org/`).

application.

To be more concrete, the ASP solver *clasp* (Gebser, Kaminski, & Schaub, 2012a; Gebser, Kaufmann, & Schaub, 2012c) represents the state of the art in solving ASP problems and has done so for several years. The Potassco team won the biennial ASP Competition in 2009, 2011, 2013 and 2014 with *clasp*. However, for each of these competitions, the team put a lot of effort into preparing *clasp*, that is, finding a well-performing parameter configuration by instantiating with more than 80 *clasp* parameters. In fact, even though *clasp* is widely used, we are not aware of any application in which it is used in its default configuration. Finding well-performing configurations requires substantial expert knowledge about the solver and problem domain. Unfortunately, in real world applications, such experts are rarely available.

There are mainly two approaches to minimize the need of human intervention to effectively set up a solver. First, *algorithm configuration* considers the problem of automatically looking for a configuration of an algorithm (Hutter, Hoos, Leyton-Brown, & Stützle, 2009). This task is performed by an algorithm configurator. Thereby, an algorithm is an abstract concept for all kinds of software; for example, *clasp* as a solver could be such an algorithm. However, an algorithm configurator needs to perform several algorithm runs with different configurations to find a well-performing configuration. On the one hand, a solver can still not be effectively used out-of-the-box because of the configuration process, which has to be done for each new instance set and often needs several days on a compute cluster. On the other hand, the requirement of a human expert is reduced and we can automatically get well-performing and specialized solvers.

Second, an orthogonal approach to algorithm configuration is the use of *algorithm portfolio solvers* that goes back to Huberman, Lukose, and Hogg (1997) for parallel portfolio solvers.[2] The idea is that not only one algorithm with a fixed configuration is used to solve a problem instance, but a portfolio of algorithms can be used to increase the chance that at least one constituent algorithm solves the instance quickly. In the simplest case, the algorithms in the portfolio run in parallel; the first algorithm that solved an instance sends a signal to terminate the other algorithms. More sophisticated approaches include running a schedule of algorithms (*algorithm scheduling*) or selecting an algorithm to be run on an instance at hand (*algorithm selection*) based on characteristics of that instance (so called instance features). All these approaches result in more robust solvers which can be effectively applied out-of-the-box to a wide range of problem instances. However, since solvers have to be robust on a large variety of instances, solvers are not as specialized as configured solvers found with algorithm configuration.

From the perspective of a solver developer, the process of software development often offers several design choices and different solving strategies. In the simplest case, a solving strategy includes an adjustable parameter, for example, the frequency of restarts. Because of interactions between these choices, decisions between the choices can not be prematurely made without loosing performance potential. The *programming by optimization* paradigm (Hoos, 2012) recommends to implement all reasonable choices and expose them as parameters. This is one of the reasons why *clasp* has more than 80 parameters nowadays and even its main developer, Benjamin Kaufmann, needs several days or weeks to manually find good configurations of *clasp* for new applications. Thus in the context of programming by optimization, a goal of this dissertation is to reduce the burden on new users of a solver but also on solver developers since

[2] An algorithm portfolio can consists of different solvers, but also of one solver with different configurations or a combination of both.

also developers are not free of wrong decisions.

This goal is achieved by providing methods to automatically build effective and robust solvers with the help of (i) algorithm configuration to automatically construct parallel portfolio solvers, (ii) algorithm schedules via ASP and (iii) algorithm selection.

One focus of this dissertation is parallel solving, because parallel solvers are increasingly key to effective solving since the advent of multi-core processors. Especially parallel portfolio-based solvers have empirically proven to be effective (for example, see Hamadi, Jabbour, and Sais (2009a) or Roussel (2011)). However, for the sequential and parallel use-case, the questions arise and are answered in this thesis: (i) how to efficiently find an effective portfolio and (ii) how to adapt more advanced portfolio techniques, such as algorithm scheduling and algorithm selection, for parallel solving.

Table 1.1 gives a first impression about the performance of our methods when applied to *clasp* on a diverse set of 1294 ASP instances.[3] Default *clasp* denotes the default configuration of *clasp*, that is, *clasp* (2.1.3) as distributed at `http://potassco.sourceforge.net/` and run out-of-the-box. We note that the performance can be gradually improved by each of the presented methods. Our best assessed approach solved 202 additional instances in comparison to *clasp*'s default by applying algorithm selection in a parallel setting using four CPU cores. We note that 82 instances were never solved by any considered *clasp* configuration in our portfolios. Hence, none of our presented methods are able to solve these instances. Nevertheless, our best approach nearly reaches this performance bound represented by a theoretical *oracle* solver that always selects the best *clasp* configuration for a given instance.

	PAR10	#Timeouts	PAR1
Sequential			
Default *clasp*	1374.18	287	176.50
Algorithm Configuration	880.55	183	116.87
Algorithm Schedule (*aspeed*)	774.72	149	152.93
Algorithm Selection (*claspfolio* 2)	497.15	101	75.66
Parallel with 4 **Cores**			
Algorithm Configuration (ACPP)	552.01	114	76.31
Algorithm Schedule (*aspeed*)	458.89	93	70.79
Algorithm Selection (*claspfolio* 2)	417.16	85	62.4
Theoretical Optimum (*oracle*)	400.17	82	57.98

Table 1.1: Performance on a diverse set of 1294 ASP instances regarding average runtime with penalized timeouts by 10 times the runtime cutoff (PAR10), number of timeouts (#TOs) and penalized average runtime with factor 1 (PAR1). Each solver had at most 600 seconds to solve an instance.

In the remainder of the chapter, we give a very brief introduction to algorithm configuration, algorithm scheduling and algorithm selection. Afterwards, we state in detail the contributions of this dissertation and preview the individual chapters.

[3]The experiment is described and discussed in detail in Chapter 7.

1.1 Introduction to Algorithm Configuration, Scheduling and Selection

Whereas algorithm configuration is the problem of finding an effective configuration of a given algorithm, algorithm scheduling and algorithm selection are portfolio-based methods to increase the robustness of an algorithm. In the following, we give a brief introduction to each of these three meta solving strategies.

1.1.1 Problem Setting

In this work, we follow the typical runtime evaluation setting for solvers for *NP*-hard problems. It is motivated by a typical user behavior: a user has a threshold on time he is willing to wait for a program's execution before he will abort the program. In the context of solvers, a solver has to solve a given problem instance within this threshold. In detail, given a set of problem instances I, an algorithm gets the chance to solve each instance $i \in I$ within a fixed runtime cutoff t_c. Typical evaluation metrics to assess the performance of algorithms are number of timeouts, penalized average runtime by factor 10 (PAR10)[4] or PAR1 (sometimes also called average runtime). Also, many solver competitions consider this setting, for example, the SAT Competitions[5] and ASP Competitions[6].

1.1.2 Algorithm Configuration

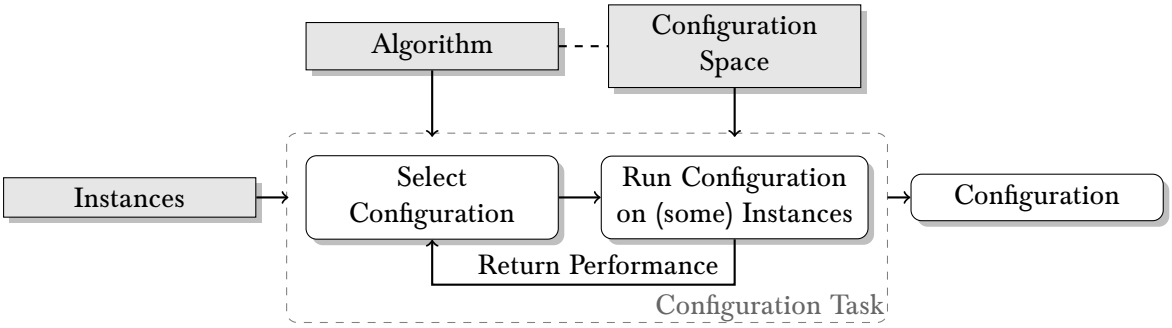


Figure 1.1: Workflow of Algorithm Configuration

Figure 1.1 shows the basic workflow of algorithm configurators. As input, the configuration task requires a set of problem instance, an algorithm, and the configuration space of the algorithm.[7] The configuration space is the cross-product of the parameters, and for discrete

[4]Penalized average runtime by factor X (PARX) is the average runtime where timeouts are penalized by X times the runtime cutoff (Hutter et al., 2009).

[5]`http://www.satcompetition.org/`

[6]`https://www.mat.unical.it/aspcomp2014/`

[7]On the technical side, more inputs for algorithm configurators are required, such as a performance metric, a configuration budget and sometimes also an initial configuration.

parameters, of exponential size in the number of parameters. In each iteration, an algorithm configurator selects a configuration from the configuration space and runs the algorithm with this configuration on one or several instances. Each algorithm run is limited by the runtime cutoff t_c. The algorithm returns its performance, for example, runtime or solution quality, to the configurator. Based on this new information, the configurator selects the next configuration to investigate. After its configuration budget is exhausted, the configurator will return the best known configuration of the algorithm. Examples of well-known algorithm configurators are *ParamILS* (Hutter et al., 2009), *GGA* (Ansótegui, Sellmann, & Tierney, 2009), *irace* (López-Ibáñez, Dubois-Lacoste, Stützle, & Birattari, 2011) and *SMAC* (Hutter, Hoos, & Leyton-Brown, 2011a).

1.1.3 Algorithm Scheduling

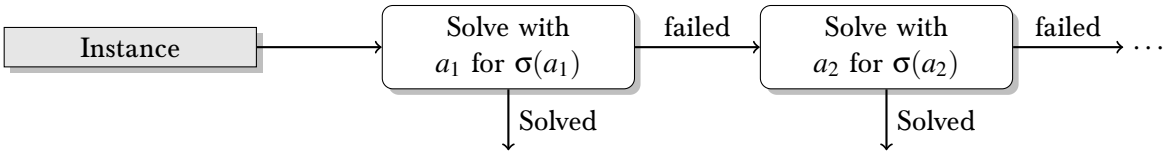


Figure 1.2: Workflow of Algorithm Schedules with algorithm $a_i \in A$ and time slices $\sigma : A \to \mathbb{R}_0^+$

Figure 1.2 shows the workflow of an algorithm schedule. Algorithm schedules are an iterative process of algorithm runs. Each algorithm a in a portfolio gets a time slice $\sigma(a)$ to solve a given instance. If the algorithms fails to do so, the next algorithm will try to solve the instance within its time slice and so on until the overall runtime cutoff t_c is reached. Therefore, the sum of all time slices has to be at most t_c. The main challenge when applying algorithm schedules is how to find the time slices for each algorithm such that the number of timeouts is minimized, and the alignment of the algorithms, that is, the sequence of algorithm runs, such that the average runtime of the schedule is minimized. For example, algorithm schedules are used in *CPhydra* (O'Mahony et al., 2008), *Fast Downward Stone Soup* (Helmert et al., 2011; Seipp, Braun, Garimort, & Helmert, 2012) and *3S* (Kadioglu, Malitsky, Sabharwal, Samulowitz, & Sellmann, 2011).

1.1.4 Algorithm Selection

Figure 1.3 shows the workflow of an algorithm selector. For a given problem instance first, numerical characteristics of this instance are computed These so-called instance features include for example, the number of variables or clauses in a SAT formula. Based on these instance features, an appropriate algorithm from a portfolio is selected to solve the given instance. The complete workflow (including feature computation, algorithm selection and running the algorithm) is limited by the runtime cutoff t_c. The main problem of applying algorithm selection is how to find a mapping from instance features to an effective algorithm for an arbitrary instance. Examples for algorithm selectors are *SATzilla* (Xu, Hutter, Hoos, & Leyton-Brown, 2008), *AQME* (Pulina & Tacchella, 2007) and *LLAMA* (Kotthoff, 2013). We note that algo-

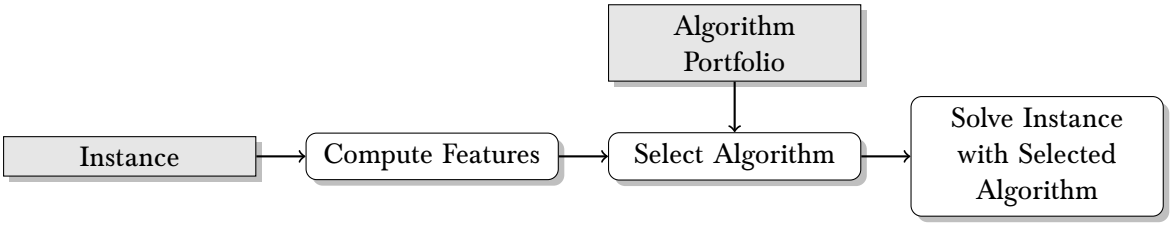


Figure 1.3: Workflow of Algorithm Selectors

rithm selectors, such as *SATzilla*, often use further techniques, such as pre-solving algorithm schedules, to increase their performance and do not solely rely on algorithm selection.

1.2 Contributions and Content

In the following, we give an overview of the content of the dissertation and point out the contributions of each individual chapter.

Chapter 2. Much time in designing a Boolean constraint solver is spent for benchmarking new ideas on large and representative benchmark sets. However, many benchmark sets are not suited for this task, for example, they include overly easy or excessively hard instances, or they are not well balanced between different types of instances. In Chapter 2, we introduce a method to select a subset of benchmark instances to efficiently and robustly benchmark Boolean constraint solvers.[8] Benchmarks on the selected subset save computational resources and thus facilitate faster development of new solvers. Additionally, a good distribution of instance characteristics leads to solvers performing more robustly on a large variety of benchmark instances.

Chapter 3. Effective parallel solvers are increasingly key to solving computationally challenging problems. Unfortunately, the manual construction of parallel solvers is non-trivial, often requiring redesign of existing sequential approaches. In Chapter 3, we study generic methods to automatically construct parallel portfolio solvers, named ACPP.[9]The ability to automatically construct parallel solvers from a sequential source reduces the burden on the developer since the development of natively parallel solvers requires special expertise, long development and debugging cycles. To reduce the amount of required computation resources, we applied the benchmark selection strategy of Chapter 2 to the empirical performance analysis of our automatically constructed parallel solvers.

Chapter 4. The rather simple approach of relying on handmade, uniform and unordered solver schedules followed by *ppfolio* (Roussel, 2011) won several medals in the 2011 SAT Competition. Inspired by this, we took advantage of the modeling and solving capacities of

[8] partly published in a conference paper (Hoos, Kaufmann, Schaub, & Schneider, 2013) – we note that Marius Lindauer (previously Marius Schneider) was leading author in all publications used in this thesis.

[9] partly published in a workshop article (Hoos, Leyton-Brown, Schaub, & Schneider, 2012)

ASP to automatically determine more refined, that is, non-uniform and ordered schedules from existing benchmarking data. In Chapter 4, we show how to efficiently model this problem using ASP.[10] Beyond the sequential case, we extend the problem to parallel schedules. We provide the problem definition as well as the ASP encoding for parallel schedules. Based on this, we introduce *aspeed*, an open-source framework that is able to efficiently find optimal algorithm schedules. Furthermore, we assess the performance of our optimized sequential and parallel schedules on several constraint solving domains and compare them with other state-of-the-art solvers. Here, we provide an interesting application for ASP, on the one hand, and on the other, a way to improve the performance and robustness of Boolean constraint solvers.

Chapter 5. Building on our award-winning, portfolio-based ASP solver *claspfolio*, we present *claspfolio* 2 in Chapter 5.[11] It provides a modular and open solver architecture that integrates several different portfolio-based algorithm selection approaches and techniques. The *claspfolio* 2 solver framework supports various feature generators, solver selection approaches, solver portfolios, as well as algorithm schedule based pre-solving techniques from Chapter 4. The default configuration of *claspfolio* 2 relies on a light-weight version of the ASP solver *clasp* to generate static and dynamic instance features. The flexible open design of *claspfolio* 2 is a distinguishing factor even beyond ASP. As such, it provides a unique framework for comparing and combining existing portfolio-based algorithm selection approaches and techniques in a single, unified framework. Taking advantage of this, we conducted an extensive experimental study to assess the impact of different feature sets, selection approaches and base solver portfolios. In addition to gaining substantial insights into the utility of the various approaches and techniques, we identified a default configuration of *claspfolio* 2 that achieves substantial performance gains not only over *clasp*'s default configuration and the earlier version of *claspfolio*, but also over manually tuned configurations of *clasp*.

Chapter 6. In practice, algorithm selection of sequential algorithms is rarely able to reach the performance of a perfect selector, that is, always selecting the best algorithm from a portfolio for a given instance. One way to improve the performance of algorithm selectors and to leverage parallel hardware architectures is the selection of a parallel per-instance portfolio of algorithms. We present *PASU*, an approach to select parallel portfolios under consideration of the uncertainty of the predicted performance of the selected algorithms, in Chapter 6. In this way, we further increase the probability to select the best-performing algorithm for a given instance. *PASU* is implemented using the flexible framework of *claspfolio* 2. We conducted an extensive experimental study to assess the utility of *PASU* on a large and diverse set of different algorithm selection scenarios from the Algorithm Selection Library (ASlib[12]).

Chapter 7. While in Chapters 3 to 6, we assessed the presented approaches on scenarios that

[10] partly published in a conference and in a journal article (Hoos, Kaminski, Schaub, & Schneider, 2012; Hoos, Kaminski, Lindauer, & Schaub, 2015). We thank Roland Kaminski for helping us to model the ASP encodings effectively.

[11] partly published in a journal paper (Hoos, Lindauer, & Schaub, 2014)

[12] `www.aslib.net`

were available at the time we developed them, in Chapter 7, we compare all of them using the same setup. Specifically, we compare parallel portfolios from the ACPP approach, algorithm schedules by *aspeed* and algorithm selection by *claspfolio* 2 with *PASU*. On the one hand, this comparison is based on a homogeneous instance set of ASP Ricochet Robots[13] instances (a transverse benchmark problem to compare different modeling and solving techniques; Gebser, Jost, Kaminski, Obermeier, Sabuncu, Schaub, and Schneider (2013)). On the other hand, we use the heterogeneous instance set of the *ASP-POTASSCO* scenario from ASlib. Based on these results, we deduce practical advice regarding the question when to apply which of our methods.

Chapter 8. In this final chapter, we discuss the presented methods in a larger context and suggest directions for future work. For example, we outline how the presented methods can be combined to create the next level of meta solving approaches.

[13] `http://en.wikipedia.org/wiki/Ricochet_Robot`

2 Robust Benchmark Set Selection for Boolean Constraint Solvers

The availability of representative sets of benchmark instances is of crucial importance for the successful development of high-performance solvers for computationally challenging problems, such as propositional satisfiability (SAT) and answer set programming (ASP). Such benchmark sets play a key role for assessing solver performance and thus for measuring the computational impact of algorithms and/or their vital parameters. On the one hand, this allows a solver developer to gain insights into the strengths and weaknesses of features of interest. On the other hand, representative benchmark instances are indispensable to empirically underpin the claims of computational benefit of novel ideas.

A representative benchmark set is composed of benchmark instances derived from a variety of different benchmark classes. Such benchmark sets have been assembled (manually) in the context of well-known solver competitions, such as the SAT and ASP competitions, and then widely used in the research literature. These sets of competition benchmarks are well-accepted, because they have been constituted by an independent committee using sensible criteria. Moreover, these sets evolve over time and thus usually reflect the capabilities (and limitations) of state-of-the-art solvers; they are also publicly available and well-documented.

However, instance sets from competitions are not always suitable for benchmarking scenarios where the same runtime cutoff is used for all instances. For example, in the last three ASP competitions, only $\approx$10% of all instances were non-trivial (runtime over 9 second, that is, 1% of the runtime cutoff) for the state-of-the-art ASP solver *clasp*, while all other instances were trivial or unsolvable for *clasp* within the time cutoff used in the competition. When benchmarking, results of benchmarks are (typically) aggregated over all instances. But if the percentage of interesting instances in the benchmark set is too small, the interesting instances have small influence on the aggregated result and the overall result is dominated by uninteresting, that is, trivial or unsolvable instances. Hence, a significant change of the runtime behaviour of a new algorithm is harder to identify on such degenerate benchmark sets. In addition, uninteresting instances unnecessarily waste computational resources and thus cause avoidable delays in the benchmarking process.

Moreover, in ASP, competition instances do not necessarily represent real world applications. In the absence of a common modelling language, benchmark instances are often formulated in the most basic common setting and thus bear no resemblance to how real world problems are addressed (for example, they are usually free of any aggregates).[1] The situation is simpler in SAT, where a wide range of benchmark instances is derived from real-world applications and are quite naturally encoded in a low-level format, without the modelling layer present in ASP. Notably, SAT competitions place considerable emphasis on a public and transparent instance

[1] In ASP competitions, this deficit is counterbalanced by a modelling track, in which each participant can use its preferred modelling language.

selection procedure (Balint, Belov, Järvisalo, & Sinz, 2012b). However, as we discuss in detail in Section 2.2, competition settings may differ from other benchmarking contexts.

In what follows, we elaborate upon the composition of representative benchmark sets for evaluating and improving the performance of Boolean constraint solvers in the context of ASP and SAT. Starting from an analysis of current practice of benchmark set selection in the context of SAT competitions (Section 2.1), we isolate a set of desiderata for representative benchmark sets (Section 2.2). For instance, sets with a large variety of instances are favourable when developing a default configuration of a solver that is desired to perform well across a wide range of instances. We rely on these desiderata for guiding the development of a parametrized benchmark selection algorithm (Section 2.3).

Overall, our approach makes use of (i) a large base set (or distribution) of benchmark instances; (ii) instance features; and (iii) a representative set of state-of-the-art solvers. Fundamentally, it constructs a benchmark set with desirable properties regarding difficulty and diversity by sampling from the given base set. It achieves diversity of the benchmark set by clustering instances based on their similarity w.r.t a given set of features, while ensuring that no cluster is overrepresented. The difficulty of the resulting set is calibrated based on the given set of solvers. Use of the benchmark sets thus obtained helps save computational resources during solver development, configuration and evaluation, while concentrating on interesting instances.

We empirically demonstrate in Section 2.4 that optimizing solvers on the obtained selection of benchmarks leads to better configurations than obtainable from the vast original set of benchmark instances. We close with a final discussion and some remarks on future work in Section 2.5.

2.1 Current Practice

The generation or selection of benchmark sets is an important factor in the empirical analysis of algorithms. Depending on the goals of the empirical study, there are various criteria for benchmark selection. For example, in the field of Boolean constraint solving, regular competitions are used to asses new approaches and techniques as well as to identify and recognize state-of-the-art solvers. Over the years, competition organizers came up with sets of rules for selecting subsets of submitted instances to assess solver performance in a fair manner. To begin with, we investigate the rules used in the well-known and widely recognized SAT Competition[2], which try to achieve (at least) three overall goals. First, the selection should be broad, that is, the selected benchmark set should contain a large variety of different kinds of instances to assess the robustness of solvers. Second, each selected instance should be significant w.r.t. the ranking obtained from the competition. Third, the selection should be fair, that is, the selected set should not be dominated by a set of instances from the same source (either a domain or a benchmark submitter).

For the 2009 SAT Competition (Berre, Roussel, & Simon, 2009) and the 2012 SAT Challenge (Balint et al., 2012b), instances were classified according to hardness, as assessed based on the runtime of a set of representative solvers. For instance, for the 2012 SAT Challenge, the organizers measured the runtimes of the best five SAT solvers from the Application and Crafted tracks of the last SAT Competition on all available instances and assigned each instance to one

[2] http://www.satcompetition.org

of the following classes: *easy* instances are solved by all solvers under 10% of the runtime cutoff, that is, 90 CPU seconds; *medium* instances are solved by all solvers under 100% of the runtime cutoff; *too hard* instances are not solved by any solver within 300% of the runtime cutoff; and *hard* instances are solved by at least one solver within 300% of the runtime cutoff but not by all solvers within 100% of the runtime cutoff. Instances were then selected with the objective to have 50% *medium* and 50% *hard* instances in the final instance set and, at the same time, to allow at most 10% of the final instance set to originate from the same source.

While the *easy* instances are assumed to be solvable by all solvers, the *too hard* instances are presumably not solvable by any solver. Hence, neither class contributes to the solution count ranking used in the competition.[3] On the other hand, *medium* instances help to rank weaker solvers and to detect performance deterioration w.r.t. previous competitions. The *hard* instances are most useful for ranking the top-performing solvers and provide both a challenge and a chance to improve state-of-the-art SAT solving.

Although using a large variety of benchmark instances is clearly desirable for robust benchmarking, the rules used in the SAT Competition are not directly applicable to our identified use cases. First, the hardness criteria and distribution used are directly influenced by the use of the solution count ranking system. On the other hand, ranking systems that also consider measured runtimes, like the careful ranking[4](van Gelder, 2011), might be better suited for differentiating solver performance. Second, limiting the number of instances from one source to achieve fairness is not needed in our setting. Furthermore, the origin of instances provides only an indirect way of achieving a heterogeneous instance set, as certain instances of different origin may in fact be more similar than other pairs of instances from the same source.

2.2 Desirable Properties of Benchmark Sets

Before diving into the details of our selection algorithm, let us first explicate the desiderata for a representative benchmark set (cf. (Hoos & Stützle, 2004)).

Large Variety of Instances. As mentioned, a large variety of instances is favourable to assess the robustness of solver performance and to reduce the risk of generalising from results that only apply to a limited class of problems. Such large variety can include different types of problems, that is, real-world applications, crafted problems, and randomly generated problems; different levels of difficulty, that is, easy, medium, and hard instances; different instance sizes; or instances with diverse structural properties. While the structure of an instance is hard to assess, a qualitative assessment could be based on visualizing the structure (Sinz, 2007), and a quantitative assessment can be performed based on instance features (Nudelman, Leyton-Brown, Hoos, Devkar, & Shoham, 2004; Xu et al., 2008). Such instance features have already proven useful in the context of algorithm selection (Xu et al., 2008; Kadioglu et al., 2011) and algorithm configuration (Hutter et al., 2011a; Kadioglu, Malitsky, Sellmann, & Tierney, 2010).

[3] Solution count ranking assesses solvers based on the number of solved instances.

[4] Careful ranking compares pairs of solvers based on statistically significant performance differences and ranks solvers based on the resulting ranking graph.

Adapted Instance Hardness. While easy problem instances are sometimes useful for investigating certain properties of specific solvers, intrinsically hard or difficult to solve problem instances are better suited to demonstrate state-of-the-art solving capabilities through benchmarking. However, in view of the nature of NP-hard problems, it is likely that many hard instances cannot be solved efficiently. Resource limitations, such as runtime cutoffs or memory limits, are commonly applied in benchmarking. Solver runs that terminated prematurely because of violations of resource limits are not helpful in differentiating solver performance. Hence, instances should be carefully selected so that such prematurely terminated runs for the given set of solvers are relatively rare. Therefore, the *distribution of instance hardness* within a given benchmark set should be adjusted based on the given resource limits and solvers under consideration. In particular, instances that are too hard (that is, for which there is a high probability of a timeout) as well as instances that are too easy, should be avoided, where hardness is assessed using a representative set of state-of-the-art solvers, as is done, for example, in the instance selection process of SAT competitions (Berre et al., 2009).

Since computational resources are typically limited, the number of benchmark instances should also be carefully calibrated. While using too few instances can bias the results, using too many instances can cost computational resources without improving the information gained from benchmarking. Therefore, we propose to start with a broad base set of instances, for example, generated by one or more (possibly parametrized) generators or a collection of previously used competition instance sets, and to select a subset of instances following our desiderata.

Free of Duplicates, Reproducible, and Publicly Available. Benchmark set should be free of duplicates, because using the same instance twice does not provide any additional information about solver performance. Nevertheless, non-trivially transformed instances can be useful for assessing the robustness of solvers (Brglez, Li, & Stallmann, 2002). To facilitate reproducibility and comparability, both the problem instances and the process of instance selection should be publicly available. Ideally, problem instances should originate from established benchmark sets and/or public benchmark libraries. To our surprise, these properties are not true for all competition sets. For example, we found duplicates in the SAT Challenge 2012, ASP Competitions 2007 and 2009 (for example, `15-puzzle.init1.gz` and `15puzzle_ins.lp.gz` in the latter).

2.3 Benchmark Set Selection

Based on our analysis of solver competitions and the resulting desiderata, we developed an instance selection algorithm. Its implementation is open source and freely available at `http://potassco.sourceforge.net`. In addition, we present a way to assess the relative robustness and quality of an instance set based on the idea of Q-scores (Balint et al., 2012b).

2.3.1 Benchmark Set Selection Algorithm

Our selection process starts from a given base set of instances I. This set can be a benchmark collection or simply a mix of previously used instances from competitions.

Inspired by past SAT competitions, a representative set of algorithms A – for example, best solvers of the last competition, the state-of-the-art (SOTA) contributors identified in the last

competition, or contributors to SOTA portfolios (Xu et al., 2012a) – is used to assess the hardness $h(i) \in \mathbb{R}$ of an instance $i \in I$. Typically, the runtime $t(i,a)$ (measured in CPU seconds) is used to assess the hardness of an instance $i \in I$ for algorithm $a \in A$. The aggregation of the runtimes of all algorithm $a \in A$ on a given instance i can be carried out in several ways, for example, by considering the minimal ($\min_{a \in A} t(i,a)$) or the average runtime ($\frac{1}{|S|} \cdot \sum_{a \in A} t(i,a)$). The resulting hardness metric is closely related to the intended ranking scheme for algorithms. For example, the minimal runtime is a lower bound of the portfolio runtime performance and represents a challenging hardness metric appropriate in the context of solution count ranking. In contrast, the average runtime would be better suited for a careful ranking (van Gelder, 2011), which uses pairwise comparisons between algorithms for each instance, because the pairs of runtimes for two algorithms are of limited value if neither of them solved the given instance within the given cutoff time. Since all algorithms contribute to the average runtime per instance, this metric will assess instances as hard even if only some solvers time out on time, and selecting instances based on it (as explained in the following) can therefore be expected to result in fewer timeouts overall.

After selecting a hardness metric, we have to choose how the instance hardness should be distributed within the benchmark set. As stated earlier, and under the assumption that the set to be created will not be used primarily in the context of solution count ranking, the performance of algorithms can be compared better, if the incidence of timeouts is minimized. This is important, for example, in the context of algorithm configuration (manual or automatic). The incidence of timeouts can be minimized by increasing the runtime cutoff, but this is infeasible or wasteful in many cases. Alternatively, we can ensure that not too many instances on which timeouts occur are selected for inclusion in our benchmark set. At the same time, as motivated previously, it is also undesirable to include too many easy instances, because they incur computational cost and, depending on the hardness metric used, can also distort final performance rankings determined on a given benchmark set.

One way to focus the selection process on the most useful instances w.r.t. hardness, namely those that are neither too easy nor too hard, is to use an appropriately chosen probability distribution to guide sampling from the given base set of instances. For example, the use of a normal (Gaussian) distribution of instance hardness in this context leads to benchmark sets consisting predominantly of instances of medium hardness, but also include some easy and hard instances. Alternatively, one could consider log-normal or exponential distributions, which induce a bias towards harder instances, as can be found in many existing benchmark sets. Compared to the instance selection approach used in SAT competitions (Balint et al., 2012b; Berre et al., 2009), this method does not require the classification of instances into somewhat arbitrary hardness classes.

The parameters of the distribution chosen for instance sampling, for example, mean and variance in the case of a normal or log-normal distribution, can be determined based on the hardness metric and runtime limit; for example, the mean could be chosen as half the cutoff time. By modifying the mean, the sampling distribution can effectively be shifted towards harder or easier benchmark instances.

As argued before, the origin of instances is typically less informative than their structure, as reflected, for example, in informative sets of instance features. Such informative sets of instance features are available for many combinatorial problems, including SAT (Xu et al., 2008), ASP (Gebser, Kaminski, Kaufmann, Schaub, Schneider, & Ziller, 2011) and CSP (O'Mahony

Algorithm 1: Benchmark Selection Algorithm

Input : instance set I; desired number of instances n; representative set of algorithms A; runtimes $t(i,a)$ with $(i,a) \in I \times A$; normalized instance features $f(i)$ for each instance $i \in I$; hardness metric $h : I \to \mathbb{R}$ of instances; desired distribution $\mathcal{D}_h$ regarding h; clustering algorithm ca; cutoff time t_c; threshold e for too easy instances;

Output: selected instances I^*

```
1  remove instances from I that are not solved by any a ∈ A within t_c;
2  remove instances from I that are solved by all a ∈ A under e% of t_c ;
3  cluster all instances i ∈ I in the normalized feature space f(i) into clusters s(i) using
   clustering algorithm ca;
4  while |I*| < n and I ≠ ∅ do
5      sample x ∈ ℝ ~ D_h;
6      select instance i* ∈ I with the nearest h(i*) to x;
7      remove i* from I;
8      if s(i*) is not over-represented then
9          add i* to I*;
10 return I*
```

et al., 2008), where they have been shown to correlate with the runtime of state-of-the-art algorithms and have been used prominently in the context of algorithm selection (see, for example, (Xu et al., 2008; Kadioglu et al., 2011)). To prevent the inclusion of too many similar instances in the benchmark sets, we cluster the instances based on their similarity in feature space. We then require that a cluster must not be over-represented in the selected instance set; in what follows, roughly reminiscent of the mechanism used in SAT competitions, we say that a cluster is over-represented if it contributes more than 10% of the instances to the final benchmark set. While other mechanisms are easily conceivable, the experiments we report later demonstrate that this simple criterion works well.

Algorithm 1 implements these ideas with the precondition that the base instance set I is free of duplicates. (This can be easily ensured by means of simple preprocessing.) In Line 1, all instances are removed from the given base set that cannot be solved by all algorithm from the representative algorithm set A within the selection runtime cutoff t_c (*rejection of too hard instances*). If solution count ranking is to be used in the benchmarking scenario under consideration, the cutoff in the instance selection process should be larger than the cutoff for benchmarking, as was done in the 2012 SAT Challenge. In Line 2, all instances are removed that are solved by all algorithms under $e\%$ of the cutoff time (*rejection of too easy instances*). For example, in the 2012 SAT Challenge (Balint et al., 2012b), all instances were removed which were solved by all algorithms under 10% of the cutoff. Line 3 performs clustering of the remaining instances based on their normlized features. To perform this clustering, the well-known k-means algorithm could be used, and the number of clusters could be computed using G-means (Hamerly & Elkan, 2003; Kadioglu et al., 2010) or by increasing the number of clusters until the clustering optimization does not improve further under a cross validation (Hill & Lewicki, 2005).

In our experiments, we used the latter, because the G-means algorithm relies on a normality assumption that is not necessarily satisfied for the instance feature data used here. Beginning with Line 4, instances are sampled within a loop until enough instances are selected or no more instances are left in the base set. To this end, $x \in \mathbb{R}$ is sampled from a distribution $\mathcal{D}_h$ induced by instance hardness metric h, such that for each sample x from hardness distribution $\mathcal{D}_h$, the instance i^* is selected whose hardness $h(i^*)$ is closest to x. Instance i^* is removed from the base instance set I. If the respective cluster $s(i^*)$ is not already over-represented in I^*, instance i^* is added to I^*, the benchmark set under construction.

2.3.2 Benchmark Set Quality

We would like to ensure that our benchmark selection algorithm produces instance sets that are in some way better than the respective base sets. At the same time, any benchmark set I^* it produces should be representative of the underlying base set I in the sense that if an algorithm performs better than a given baseline (for example, some prominent solver) on I^* it should also be better on I. However, the converse may not hold, because specific kinds of instances may dominate I but not I^*, and excellent performance on those instances can lead to a situation where an algorithm that performs better on I does not necessarily perform better on I^*.

Bayless et al. (Bayless, Tompkins, & Hoos, 2012) proposed a quantitative assessment of instance set utility. Their use case is the performance assessment of (new) algorithms on an instance set I_1 that has practical limitations, for example, the instances are too large, too hard to solve, or not enough instances are available. Therefore, a second instance set I_2 without these limitations is assessed as to whether it can be regarded as a representative proxy for the instance set I_1 during algorithm development or configuration. The key idea is that any I_2 that is a representative proxy for I_1 can be used in lieu of I_1 to assess performance of an algorithm, with the assurance that good performance on I_2 (which is easier to demonstrate or achieve) implies, at least statistically, good performance on I_1.

To assess the utility of an instance set, they use algorithm configuration (Hutter et al., 2009, 2011a; Kadioglu et al., 2010). An algorithm configurator is used to find a configuration $c := a(c_I)$ of algorithm a on instance set I by optimizing, for example, the runtime of a. If I_2 is a representative proxy for I_1, the algorithm configuration $a(c_{I_2})$ should perform on I_1 as well as a configuration optimized directly on I_1, that is, $a(c_{I_1})$. The Q-score $Q(I_1, I_2, a, m)$ defined in Equation (2.1) is the performance ratio of $a(c_{I_1})$ and $a(c_{I_2})$ on I_1 with respect to a given performance metric m. A large Q-score means I_2 is a good proxy for I_1. The short form of $Q(I_1, I_2, a, m)$ is $Q_{I_1}(I_2)$.

To compare both sets, I_1 and I_2, we want to know whether I_2 is a better proxy for I_1 than vice versa. To this end, we extended the idea in (Bayless et al., 2012) and propose the Q*-score of I_1 and I_2 by computing the ratio of $Q_{I_1}(I_2)$ and $Q_{I_2}(I_1)$ as per Equation (2.2). If I_1 is a better proxy for I_2 than vice versa, the Q*-score $Q^*(I_1, I_2)$ is larger than 1.

$$Q(I_1, I_2, a, m) = \frac{m(a(c_{I_1}), I_1)}{m(a(c_{I_2}), I_1)} \tag{2.1}$$

$$Q^*(I_1, I_2) = \frac{Q_{I_1}(I_2)}{Q_{I_2}(I_1)} \tag{2.2}$$

We use the Q*-score to assess the quality of the sets I^* obtained from our benchmark selection algorithm in comparison to the respective base sets I. Based on this score, we can assess the degree to which our benchmark selection algorithm succeeded in producing a set that is representative of the given base set in the way motivated earlier. Thereby, a Q^*-score ($Q^*(I_1 = I, I_2 = I^*)$) and a Q-score ($Q_{I_1=I}(I_2 = I^*)$) of larger than 1.0 indicates that I^* is better proxy for I than vice versa and I^* is a good proxy for I.

2.4 Empirical Performance Analysis

We evaluated our benchmark set selection approach by means of the Q^*-score criterion on widely studied instance sets from SAT and ASP competitions.

Instance Sets. We used three base instance sets to select our benchmark set: *SAT-Application* includes all instances of the *application* tracks from the 2009 and 2011 SAT Competition and 2012 SAT Challenge; *SAT-Crafted* includes instances of the *crafted* tracks (resp. hard combinatorial track) of the same competitions; and *ASP-Set* includes all instances of the 2007 ASP Competition (SLparse track), the 2009 ASP Competition (with the encodings of the Potassco group (Gebser, Kaminski, Kaufmann, Ostrowski, Schaub, & Schneider, 2011)), the 2011 ASP Competition (decision NP-problems from the system track), and several instances from the ASP benchmark collection platform *asparagus*[5]. Duplicates were removed from all sets, resulting in 649 instances in *SAT-Application*, 850 instances in *SAT-Crafted*, and 2589 instances in *ASP-Set*.

Solvers. In the context of the two sets of SAT instances, the best two solvers of the application track, that is, *Glucose* (Audemard & Simon, 2012) (2.1) and *SINN* (Yasumoto, 2012), and of the hard combinatorial track, that is, *clasp* (Gebser et al., 2011) (2.0.6) and *Lingeling* (Biere, 2012) (agm), and the best solver of the random track, that is, *CCASAT* (Cai, Luo, & Su, 2012), of the 2012 SAT Challenge were chosen as representative state-of-the-art SAT solvers. *clasp* (Gebser et al., 2011) (2.0.6), *cmodels* (Giunchiglia, Lierler, & Maratea, 2006) (3.81) and *smodels* (Simons, Niemelä, & Soininen, 2002) (2.34) were selected as competitive and representative ASP solvers capable of reading the *smodels*-input format (Syrjänen, 2001).

Instance Features. We used efficiently computable, structural features to cluster instances. The fifty-four *base* features of the feature extractor of *SATzilla* (Xu et al., 2008) (2012) were utilized for SAT. The seven structural features of *claspfolio* (Gebser et al., 2011) were considered for ASP, namely, tightness (0 or 1), number of atoms, all rules, basic rules, constraint rules, choice rules, and weight rules of the grounded program. For feature computation, a runtime limit of 900 CPU seconds per instance and a z-score normalization was used. Any instance for which the complete set of features could not be computed within 900 seconds was removed from the set of candidate instances. This led to the removal of 52 instances from the *SAT-Application* set, 2 from the *SAT-Crafted* set, and 3 from the *ASP-Set* set.

[5] http://asparagus.cs.uni-potsdam.de/

Execution Environment and Solver Settings. All our experiments were performed on a computer cluster with dual Intel Xeon E5520 quad-core processors (2.26 GHz, 8192 KB cache) and 48 GB RAM per node, running Scientific Linux (2.6.18-308.4.1.el5). Each solver run was limited to a runtime cutoff of 900 CPU seconds. Furthermore, we set parameter e in our benchmark selection procedure to 10, that is, instances solved by all solvers within 90 CPU seconds were discarded, and the number of instances to select (n) to 200 for SAT (because of the relatively small base sets) and 300 for ASP. After filtering out *too hard* instances (Line 1 of Algorithm 1), 404 instances remained in *SAT-Application*, 506 instances in *SAT-Crafted* and 2190 instances in *ASP-Set*; after filtering out *too easy* instances (Line 2), we obtained sets of size 393, 425, and 1431, respectively.

Clustering. To cluster the instances based on their features (Line 3), we applied k-means 100 times with different randomised initial cluster centroids. To find the optimal number of clusters, we gradually increased the number of clusters (starting with 2) until the quality of the clustering, assessed via 10-fold cross validation and 10 randomised repetitions of k-means for each fold, did not improve any further (Hill & Lewicki, 2005). This resulted in 13 clusters for each of the two SAT sets, and 25 clusters for the ASP set.

Selection. To measure the hardness of a given problem instance, we used the average runtime over all representative solvers. We considered a cluster to be over-represented (Line 8) if more than 20% of the final set size (n) were selected for SAT, and more than 5% in case of ASP; the difference in threshold was motivated by the fact that substantially more clusters were obtained for the *ASP-Set* set than for *SAT-Application* and *SAT-Crafted*.

Algorithm Configuration. After generating the benchmark sets *SAT-Application**, *SAT-Crafted** and *ASP-Set** using our automated selection procedure, these sets were evaluated by assessing their Q^*-scores. To this end, we used the freely available, state-of-the-art algorithm configurator *ParamILS* (Hutter et al., 2009) to configure the SAT and ASP solver *clasp* (2.0.6). *clasp* is a competitive solver in several areas of Boolean constraint solving[6] that is highly parameterized, exposing 46 performance-relevant parameters for SAT and 51 for ASP. This makes it particularly well suited as a target for automated algorithm configuration methods and hence for evaluating our instance sets. Following standard practice, for each set, we performed 10 independent runs of *ParamILS* of 2 CPU days each and selected from these the configuration with the best training performance as the final result of the configuration process for each instance set.

Sampling Distributions. One of the main input parameters of Algorithm 1 is the sampling distribution. With the help of our Q^*-score criterion, three distributions are assessed: a normal (Gaussian) distribution, a log-normal distribution, and an exponential distribution. The parameters of these distributions were set to the empirical statistics (for example, empirical mean and variance) of the hardness distribution over the base sets. The log-normal and exponential distributions have fat right tails and typically reflect better the runtime behaviour of solvers for NP problems than the normal distribution. However, when using the average runtime as our

[6] *clasp* won several first places in previous SAT, PB and ASP competitions.

Distribution	PAR10 on I			PAR10 on I^*			Q*-score
	c_{def}	c_I	c_{I^*}	c_{def}	c_I	c_{I^*}	
SAT-Application							
Normal	4629	4162	3997	3410	2667	1907	1.46
Log-Normal	4629	4162	4683	3875	2601	3487	0.66
Exponential	4629	4162	4192	2969	2380	2188	1.08
SAT-Crafted							
Normal	5226	5120	5056	2429	2155	1752	1.25
Log-Normal	5226	5120	5184	3359	3235	3184	1.04
Exponential	5226	5120	5072	1958	1819	1523	1.21
ASP-Set							
Normal	2496	1239	1072	1657	705	557	1.46
Log-Normal	2496	1239	1128	3136	1173	678	1.90
Exponential	2496	1239	1324	1648	710	555	1.20

Table 2.1: Comparison of set qualities of the base sets I and benchmark sets I^* generated by Algorithm 1; evaluated with Q^*-Scores with $I_1 = I$, $I_2 = I^*$, *clasp* as algorithm A and PAR10-scores as performance metric m

hardness metric, the instances sampled using a normal distribution are not necessarily atypically easy. For instance, an instance i, on which half of the representative solvers have a timeout while the other half solve the instance in nearly no time, has an average runtime of half of the runtime cutoff. Therefore, the instance is medium hard and will be likely selected by using the normal distribution.

In Table 2.1, we compare the benchmark sets we obtained from the base sets *SAT-Application*, *SAT-Crafted* and *ASP-Set* when using these three types of distributions, based on their Q^*-scores. On the left of the table, we show the PAR10 performance on the base set I of the default configuration of *clasp* (c_{def}; we use this as a baseline), the configuration c_I found on the base set I, and the configuration c_{I^*} found on the selected set I^*; this is followed by the performance on the benchmark sets I^* generated using our new algorithm. The last column reports the Q^*-score values for the pairs of sets I and I^*.

For all three instance sets, the Q^*-scores obtained via the normal distribution were larger than 1.0, indicating that c_{I^*} performed better than c_I and the set obtained from our benchmark selection algorithm I^* proved to be a good alternative to the entire base set I. Although on the *ASP-Set* set, by using the log-normal distribution a larger Q^*-score (1.90) was obtained than for the normal distribution (1.46), on the *SAT-Application* set, using the log-normal distribution did not produce good benchmark sets. When using exponential distributions, Q^*-scores are larger than 1.0 in all three cases, but smaller than those obtained with normal distributions.

When using the normal distribution, configuration c_{I^*} performed better than c_I on both sets I and I^* (implying $Q_{I_1}(I_2) > 1.0$). Therefore, configuration on the selected set I^* leads to faster (and more robust) configurations than on the base set I. Furthermore, the benchmark sets produced by our algorithm are smaller and easier than the respective base sets. Hence, less

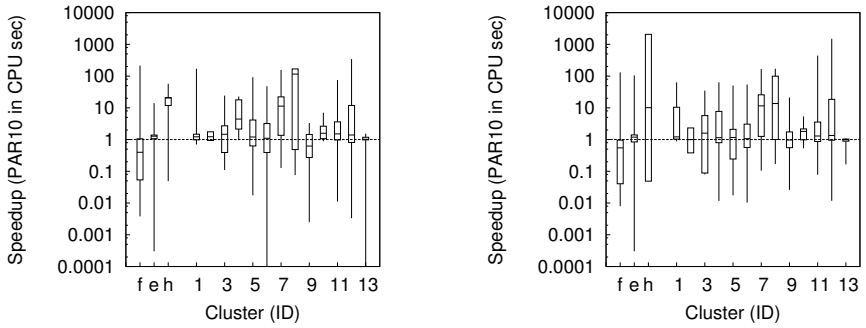


Figure 2.1: Boxplots indicating the median, quartiles minimum and maximum speedup achieved on the instance clusters within the base set *SAT-Application*; (left) compares $c_{default}$ and c_I (high values are favourable for c_I); (right) compares $c_{default}$ and c_{I^*} (high values are favourable for c_{I^*}); special clusters: s_f uncompleted feature computation; s_e too easy, s_h too hard;

CPU time is necessary to assess the performance of an algorithm on those benchmark sets. For instance, the default configuration of *clasp* needed 215 CPU hours on the base *ASP-Set* set and only 25 CPU hours on the benchmark set *ASP-Set**. For developing a new algorithm or configuring an algorithm (manually or automatically), fast and informative assessment, as facilitated by our new benchmark set generation algorithm, is very important.

Cluster Assessment. An additional advantage of Algorithm 1 is the fact that it produces a feature-based instance clustering, which can be further used to assess more precisely the performance of algorithms (or configurations). Normally, the performance of an algorithm is assessed over an entire instance set, but with the help of instance clusters, the performance can be assessed on different types of instances. This is useful, for example, in the context of developing a robust solver which should perform equally well across different types of instances. An example for such a solver is the CPLEX solver for mixed integer programming (MIP) problems, which is designed to perform well over a broad range of application contexts, each of which gives rise to different types of MIP instances.

The box plots in Figure 2.1 show the speedups (y-axis) of the configurations c_I (left) and c_{I^*} (right; while sampling with a normal distribution) against the default configuration c_{def} of *clasp* on each cluster $s_{1..13}$ (x-axis) within the *SAT-Application* base set. Furthermore, three special clusters contain the instances that were discarded in Algorithm 1 because, feature computation could not be completed (s_f), they were too easy (s_e), or too hard (s_h).

The comparison against a common baseline, here: the default configuration, helps to determine whether the new algorithm improved only on some types of instance or on all. For instance, configuration c_I (configured on the base set; left plot) improved the performance by two orders of magnitude on cluster s_8 but is slightly slower on s_9. However, configuration c_{I^*} (configured on the set generated by Algorithm 1; right plot) achieved better median performance on all clusters except for s_f. In addition, the comparison between both plots reveals that c_{I^*} produces fewer outliers than c_I, especially on clusters s_6, s_9, s_{11} and S_{13}. Similar results (not shown here) were obtained for *SAT-Crafted* and *ASP-Set*. Therefore, c_{I^*} can be considered to be

a more robust improvement over c_{def} than c_I.

We believe that the reason for the robustness of configuration c_{I^*} lies in the fact that the (automatic) configuration process tends to be biased by instance types that are highly represented in a given training set. Since Algorithm 1 produces sets I^* that cover instance clusters more evenly than the respective base sets I, the configuration process is naturally guided more towards robust performance improvements across all clusters.

Particular attention should be paid to the special clusters s_f, s_e and s_h for the assessment of c_{I^*}, because the instances contained in these clusters are not at all represented in I^*. On none of our experiments with the three types of sampling distributions did we ever observe that the performance of c_{I^*} on the *too hard* instances s_h decreased; in fact, it sometimes increased. In contrast, the performance on the *too easy* instances s_e and instances with no features s_f was less consistent, and we observed speedups between 300 and 0.1 in comparison to c_I. Therefore, the threshold for filtering too easy instances e should be set conservatively (below 10%), to ensure that not too many too easy instances are discarded (we note that this is in contrast to common practice in SAT competitions).

Furthermore, our Algorithm 1 ensures that no cluster is over-represented, but does not ensure a sufficient representation of all clusters in the selected set. For instance, cluster s_4 has 141 instances in the base *ASP-Set* set but only one instance in *ASP-Set** set (with normal distribution). Nevertheless, a low representation of a cluster in the selected set did not necessarily harm the configuration process, and in most observed cases, the configuration c_{I^*} performed as well as c_I on the under-represented clusters.

2.5 Conclusion

In this work, we have introduced an algorithm for selecting instances from a base set or distribution to form an effective and efficient benchmark set. We consider a benchmark set to be effective, if a solver configured on it performs at least as well as when configured on the original set, and we consider it to be efficient, if the instances in it are on average easier to solve than those in the base set. By using such benchmark sets, the computational resources required for assessing the performance of a solver can be reduced substantially. Our benchmark selection procedure can use arbitrary sampling distributions; yet, in our experiments, we found that using a normal (Gaussian) distribution is particularly effective. Since our approach filters out instances considered too easy or too hard for the solver under consideration, it can lead to a situation where the performance of a given solver, when configured on the benchmark set, becomes worse on those discarded instances. However, the risk of worsening the performance on too hard instances can be reduced by setting the runtime cutoff of the selection process higher than in the actual benchmark. Then, the selected set contains very challenging instances under the runtime cutoff in the benchmark, which are yet known to be solvable. We have also demonstrated that clustering of instances based on instance features facilitates diagnostic assessments of the degree to which a solver performs well on specific types of instances or across an entire, heterogeneous benchmark set. Our work reported here is primarily motivated by the desire to develop solvers that perform robustly well across a wide range of problem instances, as has been (and continues to be) the focus in developing solvers for many hard combinatorial problems.

In future work, it may be interesting to ensure that semantically different types of instances, such as satisfiable and unsatisfiable instances in the case of SAT, are represented evenly or equivalently as in a given base set. Furthermore, one could consider more sophisticated ways to assess the over-representation of feature-based clusters and to automatically adjust the sampling process based on the number of clusters and their sizes. Finally, we believe that it would be interesting to study criteria for assessing the robustness of solver performance across clusters and to use such criteria for automatic algorithm configuration.

3 Automatic Construction of Parallel Portfolios via Algorithm Configuration

Over most of the last decade, additional computational power has come primarily in the form of increased parallelism. As a consequence, effective parallel solvers are increasingly key to solving computationally challenging problems. Unfortunately, the manual construction of parallel solvers is non-trivial, often requiring fundamental redesign of existing, sequential approaches, as stated in (Hamadi & Wintersteiger, 2012) as the challenge of *Starting from Scratch.* It is thus very appealing to employ generic methods for the construction of parallel solvers from inherently sequential sources as a first step. Indeed, the prospect of a substantial reduction in human development cost means that such approaches can have a significant impact, even if their performance does not reach that of special-purpose parallel designs—just as high-level programming languages are useful, even though compiled software tends to fall short of the performance that can be obtained from expert-level programming in assembly language. One promising approach for parallelizing sequential algorithms is the design of parallel algorithm portfolios (Huberman et al., 1997; Gomes & Selman, 2001).

In this work[1], we study generic methods for solving a problem we call Automatic Construction of Parallel Portfolios (ACPP), that is, how can we construct automatically a parallel solver from a sequential solver or a set of sequential solvers. This task can be understood as falling within the *programming by optimization* paradigm (Hoos, 2012) in that it involves the design of software in which many design decisions have been deliberately left open during the development process (here exposed as parameters of SAT solvers) to be made automatically later (here by means of an automated algorithm configurator) in order to obtain optimized performance for specific use cases. Hence, the only requirement to apply our ACPP methods is a sequential solver with a rich and complementary configuration space.

We study three variants of the ACPP problem. First, we consider building parallel portfolios starting from a single, highly parametric sequential solver design. However, for well-studied problems (for example, SAT), there often exist a wide range of different solvers that contribute to the state of the art (see, for example, (Xu et al., 2012a)). Complementarities among such solvers can be exploited by algorithm portfolios, whether driven by algorithm selection (like SATzilla (Xu et al., 2008)) or by parallel execution (such as *ppfolio* (Roussel, 2011) or *pfolioUZK* (Wotzlaw, van der Grinten, Speckenmeyer, & Porschen, 2012)). Thus, the second problem we consider is leveraging such complementarities within the context of the ACPP problem, generating a parallel portfolio based on a design space induced from a set of multiple (possibly parametrized) solvers. Finally, some parallel solvers already exist; these have the advantage that they can increase performance by communicating intermediate results—notably, learned clauses—between different processes. The third problem we study is constructing parallel portfolios from a set containing both sequential and parallel solvers.

[1] which extends a previous workshop publication (Hoos et al., 2012)

We investigate four methods for solving the ACPP problem.

1. *Global* simultaneously configures all solvers in a k-solver parallel portfolio, representing this ACPP problem as a single-algorithm configuration problem with a design space corresponding to the kth Cartesian power of the design space of the given sequential solver. This has the advantages of simplicity and comprehensiveness (no candidate portfolios are omitted from the design space) but the disadvantage that the size of the design space increases exponentially with k, which quickly produces extremely difficult configuration problems.

2. *hydra* is a method for building portfolio-based algorithm selectors from a single, highly parameterized solver (Xu, Hoos, & Leyton-Brown, 2010). It proceeds iteratively. In the first round, it aims to find a configuration that maximizes overall performance on the given dataset. In the $i+1$st round, it aims to find a configuration that maximizes marginal contribution across the configurations identified in the previous i rounds. In the original version of *hydra*, these marginal contributions were calculated relative to the current selector; in the latest version of *hydra*, they are determined based on an idealized, perfect selector (Hutter, Hoos, & Leyton-Brown, 2014). The wall-clock performance of a perfect selector across i solvers (also known as *virtual best solver*) is of course the same as the wall-clock performance of the same i solvers running in parallel; thus, the same general idea can be used to build parallel portfolios. (Building a parallel portfolio in this way has the added advantage that no instance features are required, since there is no need to select among algorithms.) We introduce some enhancements to this approach for the parallel portfolio setting (discussed in Section 3.2.1.3), and refer to our method as *parHydra*.

3. *isac* is a second method for building portfolio-based algorithm selectors from parallel portfolios (Kadioglu et al., 2010), which works by clustering instances based on instance features and configuring a different solver for each cluster. Like *hydra*, *isac* can be adapted to the parallel setting. Because our implementation of this idea differs somewhat from *isac*—chiefly in its reliance on the underlying clustering algorithm and in feature normalization; see Section 3.2.1.4—we refer to our parallel variant as *Clustering*.

4. Some parallel solvers only achieve strong performance when running on more than one core; such solvers will not be selected by a greedy approach like *parHydra*. To overcome this problem, we introduce a new method called *parHydra*$_b$, which augments *parHydra* to train b solvers per iteration. This method trades off the computational benefit of *parHydra*'s greedy approach with the greater coverage of *Global*.

We evaluated our ACPP methods on SAT. We chose this domain because it is highly relevant to academia and industry and has been widely studied. We thus had access to a wide range of strong, highly parametric solvers and were assured that the bar for demonstrating efficacy of parallelization strategies was appropriately high. We note that our approach is not limited to SAT solvers and can be directly applied to other domains. To evaluate our methods in the single-solver setting, we studied both *Lingeling* and *clasp*, prominent, highly parametric state-of-the-art solvers for SAT. *Lingeling* won a gold medal in the application (wall-clock) track of the 2011 SAT Competition and *clasp* placed first in the hard combinatorial track of the 2012

SAT Challenge. For the generation of parallel portfolios involving multiple solvers, we took the solvers from *pfolioUZK*, a parallel portfolio solver based on several solvers in their default configurations that won the gold medal in the parallel track of the 2012 SAT Challenge. This set includes *Plingeling*, a parallel solver.

Our results demonstrate that *parHydra* works well and robustly for the task of producing parallel portfolios based on a single solver. Its performance on standard 8-core CPUs compared favourably with that of hand-crafted parallel SAT solvers. For the generation of parallel algorithm portfolios based on a set of both parallel and sequential solvers, we found that $parHydra_b$ performed best, even better than *pfolioUZK*. More detailed experimental results and open-source code are available at `http://www.cs.uni-potsdam.de/acpp`.

3.1 Related Work

Well before widespread interest in multi-core computing, the potential benefits of parallel algorithm portfolios were identified in seminal work by Huberman et al. (Huberman et al., 1997). Gomes et al. (Gomes & Selman, 2001) further investigated conditions under which such portfolios outperform their component solvers. Both lines of work considered prominent constraint programming problems (graph colouring and quasigroup completion), but neither presented methods for automatically constructing portfolio solvers. Instead, portfolios of algorithms first saw practical application in this domain as the basis for algorithm selectors such as SATzilla (Xu et al., 2008) and subsequently with a wide range of additional methods (see, for example, (Kotthoff, 2012)). Parallel portfolios have also seen practical impact, both in cases where the allocation of computational resources to algorithms in the portfolio is static (Petrik & Zilberstein, 2006; Yun & Epstein, 2012) and where a portfolio's constituent algorithms can change over time (Gagliolo & Schmidhuber, 2006). In the field of SAT solving, *3Spar* (Malitsky, Sabharwal, Samulowitz, & Sellmann, 2012) and *CSCHpar* (Malitsky, Sabharwal, Samulowitz, & Sellmann, 2013b, 2013a) introduced selection of parallel portfolios. All of these methods, whether parallel or selection-based, build a portfolio from a relatively small candidate set of distinct algorithms. While, in principle, these methods could also be applied given a set of algorithms expressed implicitly as the configurations of one parametric solver, in practice, they are useful only when the set of candidates is relatively small. The same limitation applies to existing approaches that combine algorithm selection and scheduling, notably *CPhydra* (O'Mahony et al., 2008), which also relies on cheaply computable features of the problem instances to be solved and selects multiple solvers to be run in parallel, and *aspeed* (Hoos et al., 2012), which computes (parallel) algorithm schedules by taking advantage of the modelling and solving capacities of Answer Set Programming (ASP (Baral, 2003; Gebser et al., 2012)).

Recently, automatic algorithm configuration has become increasingly effective, with the advent of high-performance methods such as *ParamILS* (Hutter et al., 2009), *GGA* (Ansótegui et al., 2009), *irace* (López-Ibáñez et al., 2011) and *SMAC* (Hutter et al., 2011a). As a result, there has been recent interest in automatically identifying useful portfolios of configurations from large algorithm design spaces. As before, such portfolio-construction techniques were first demonstrated to be practical in the case of portfolio-based algorithm selectors. We have already discussed the two key methods for solving this problem: *hydra* (Xu et al., 2010) greedily constructs a portfolio by configuring solvers iteratively, changing the configurator's objec-

tive function at each iteration to direct it to maximize marginal contribution to the portfolio; *isac* (Kadioglu et al., 2010) clusters instances based on features and runs the configurator separately for each cluster. In (Malitsky & Sellmann, 2012), Malitsky et al. extended scope of *isac* to the construction of portfolios from a set of different solvers. However, there are three differences between the construction of sequential portfolios and static parallel portfolios: (i) the size of the portfolio is unlimited in the sequential case and limited to the number of used processor cores in the parallel case; (ii) a sequential portfolio solver has to select somehow component solvers which can result in wrong decision; static parallel solvers ran the entire portfolio in parallel and perform nearly as good as the virtual best solver of this portfolio; (iii) using several cores in parallel induces hardware caused overhead which has to be considered in the configuration process.

Parallel SAT solvers have received increasing attention in recent years. *ManySAT* (Hamadi et al., 2009a; Hamadi, Jabbour, & Sais, 2009b; Guo, Hamadi, Jabbour, & Sais, 2010) was one of the first parallel SAT solvers. It is a static portfolio solver that uses clause sharing between its components, each of which is a manually configured, DPLL-type SAT solver based on *MiniSat* (Eén & Sörensson, 2004). *PeneLoPe* (Audemard, Hoessen, Jabbour, Lagniez, & Piette, 2012) is based on *ManySAT* and adds several policies for importing and exporting of clauses between the threads. *Plingeling* (Biere, 2010, 2011) is based on a similar design; its version 587, which won a gold medal in the application track of the 2011 SAT Competition (with respect to wall clock time on SAT+UNSAT instances), and the 2012 version *ala*, share unit clauses as well as equivalences between its component solvers. Similarly, *CryptoMiniSat* (Soos, Nohl, & Castelluccia, 2009), which won silver in the application track of the 2011 SAT Competition, shares unit and binary clauses. *clasp* (Gebser et al., 2012c) is a state-of-the-art solver for SAT, ASP and PB that supports parallel multithreading (since version 2.0.0) for search space splitting and/or competing strategies, both combinable with a portfolio approach. *clasp* shares unary, binary and ternary clauses, and (optionally) offers a parameterized mechanism for distributing and integrating (longer) clauses. Finally, *ppfolio* (Roussel, 2011) is a simple, static parallel portfolio solver for SAT without clause sharing that uses *CryptoMiniSat*, *Lingeling*, *clasp*, *tnm* (Wei & Li, 2009) and *march_hi* (Heule, Dufour, van Zwieten, & van Maaren, 2004) in their default configurations as component solvers, and that won numerous medals at the 2011 SAT Competition. Like the previously mentioned portfolio solvers for SAT, *ppfolio* was constructed manually, but uses a very diverse set of high-performance solvers as its components. *pfolioUZK* (Wotzlaw et al., 2012) follows the same idea as used for *ppfolio* but uses other component solvers; it won the parallel track of the 2012 SAT Challenge. On one hand, ACPP can be understood as automatically replicating the (hand-tuned) success of solvers like *ManySAT*, *Plingeling*, *CryptoMiniSat* or *clasp*, which are inherently based on different configurations of a single parametric solver; on the other, it is also concerned with automatically producing effective parallel portfolio from multiple solvers, such as *ppfolio* and *pfolioUZK*, while exploiting the rich design spaces of these component solvers.

3.2 Parallel Portfolio Configuration from a Single Sequential Solver

We begin by considering the problem of automatically producing a parallel portfolio solver from a single, highly-parametric sequential solver; this closely resembles the problem (manually) addressed by the developers of solvers like *ManySAT*, *Plingeling*, *CryptoMiniSat* and *clasp*. First of all, we define our three ACPP methods. Then, we show exemplarily how well our ACPP portfolio solvers perform based on *Lingeling* and *clasp*. Also, the empirical scalability of our trained ACPP solvers is analysed. In case of availability of clause sharing, we extend our ACPP solvers with clause sharing and investigate how much the performance can be improved further.

3.2.1 Approach

We now describe three methods automatically constructing parallel portfolios from a single parametric solver. We first introduce formal notation and then define our methods.

3.2.1.1 Formal Notation

We use C to denote the configuration space of our parametric solver, $c \in C$ to represent individual configurations, and I to refer to the given set of problem instances. Our goal is to optimize (without loss of generality, to minimize) performance according to a given metric m. (In our experiments, we minimize penalized average runtime, PAR10.[2]) We use a k-tuple $c_{1:k} = (c_1, \ldots, c_k)$ to denote a parallel portfolio with k component solvers. The parallel portfolio's full configuration space is $C^k = \prod_{l=1}^{k} \{(c) \mid c \in C\}$, where the product of two configuration spaces X and Y is defined as $\{x||y \mid x \in X, y \in Y\}$, with $x||y$ denoting the concatenation (rather than nesting) of tuples. Let AC denote a generic algorithm configuration procedure. (In our experiments, we used *SMAC* (Hutter et al., 2011a)). Following established best practices (see (Hutter et al., 2011a)), we performed n independent runs of AC, obtained configured solvers $c^{(j)}$ with $j \in \{1 \ldots n\}$ and kept the configured solver $\hat{c}$ with the best performance on instance set I according to metric m. By t_b we denote the overall time budget available for producing a parallel portfolio solver.

3.2.1.2 Simultaneous configuration of all component solvers (*Global*)

Our first portfolio configuration method is the straightforward extension of standard algorithm configuration to the construction of a parallel portfolio (see Algorithm 2). Specifically, if the given solver has ℓ parameters, we treat the portfolio $c_{1:k}$ as a single algorithm with $\ell \cdot k$ parameters inducing a configuration space of size $|C|^k$, and configure it directly. As noted above, we identify a single configuration as the best of n independent runs of AC. These runs can be performed in parallel, meaning that this procedure requires wall clock time t_b/n if n machines with k cores are available. The practicality of this approach is limited by the fact that the global configuration space C^k to which AC is applied grows exponentially with k. However, given a powerful configurator, a moderate value of k and a reasonably sized C, this simple approach

[2] PARX penalizes each timeout with X times the given cutoff time (Hutter et al., 2009).

Algorithm 2: Portfolio Configuration Procedure *Global*

Input : parametric solver with configuration space C; desired number k of component solvers; instance set I; performance metric m; configurator AC; number n of independent configurator runs; total configuration time budget t_b

Output: parallel portfolio solver with portfolio $\hat{c}_{1:k}$

1 **for** $j := 1 \ldots n$ **do**

2 obtain portfolio $c^{(j)}_{1:k}$ by running AC on configuration space $\prod_{l=1}^{k}\{(c) \mid c \in C\}$ on I using m for time t_b/n

3 choose $\hat{c}_{1:k} \in \arg\min_{c^{(j)}_{1:k} | j \in \{1 \ldots n\}} m(c^{(j)}_{1:k}, I)$ that achieved best performance on I according to m

4 **return** $\hat{c}_{1:k}$

has the potential to be effective, especially when compared to the manual construction of a parallel portfolio.

3.2.1.3 Iterative configuration of component solvers (*parHydra*)

The key problem with *Global* is that C^k may be so large that AC cannot effectively search it. We thus consider an extension of the *hydra* methodology to the ACPP problem, which we dub *parHydra* (see Algorithm 3). This method has the advantage that it adds and configures component solvers one at a time. The key idea is to use AC only to configure the component solver added in the given iteration, leaving all other components clamped to the configurations that were determined for them in previous iterations. The procedure is greedy in the sense that in each iteration i, it attempts to add a component solver to the given portfolio $\hat{c}_{1:i-1}$ in a way that myopically optimizes the performance of the new portfolio $\hat{c}_{1:i}$ (Line 4). While the sets of n independent configurator runs in Line 2 can be performed in parallel (as in *Global*), the choice of the best-performing configuration $\hat{c}_{1:i}$ has to be made after each iteration i, introducing a modest overhead compared to the cost of the actual configuration runs.

A disadvantage of the original *hydra* approach is that it discards any intermediate results learned during configuration when it proceeds to the next iteration. In particular, configurations that were examined but not selected may turn out to be useful later on. We thus introduce a new idea here—which, indeed, can also be applied to the construction of portfolio-based algorithm selectors—as follows. We identify the unselected configuration $c^{(j)} \neq \hat{c}_{i:i}$ with the best marginal contribution to the current portfolio $\hat{c}_{1:i}$ (Line 5), and use it to initialize the configuration procedure in the next iteration (Line 3). Our intention is that using different initial configurations in each iteration will more quickly guide the configuration procedure to complementary parts of the configuration space.

Another way that *parHydra* differs from the original *hydra* methodology is that it runs entire portfolios on each instance considered during configuration. Because we target multicore machines, we consider these computational resources to be available without cost. While *hydra* explicitly modifies the performance metric in each round, *parHydra* thus achieves the same modification implicitly, optimizing marginal contribution to the existing portfolio because only the ith element of the portfolio is available to be configured in the ith iteration. Because *parHy-*

Algorithm 3: Portfolio Configuration Procedure *parHydra*

Input : parametric solver with configuration space C; desired number k of component solvers; instance set I; performance metric m; configurator AC; number n of independent configurator runs; total configuration time t_b

Output: parallel portfolio solver with portfolio $\hat{c}_{1:k}$

```
for i := 1...k do
    for j := 1...n do
        obtain portfolio c^(j)_{1:i} := ĉ_{1:i-1}||c^(j) by running AC on configuration space
        {ĉ_{1:i-1}} × {(c) | c ∈ C} and initial incumbent ĉ_{1:i-1}||c_init on I using m for time
        t_b/(k·n)
    let ĉ_{1:i} ∈ arg min_{c^(j)_{1:i} | j∈{1...n}} m(c^(j)_{1:i}, I) be the configuration which achieved best
    performance on I according to m
    let c_init ∈ arg min_{c^(j) | j∈{1...n}} m(ĉ_{1:i}||c^(j), I) be the configuration that has the largest
    marginal contribution to ĉ_{1:i}
return ĉ_{1:k}
```

dra only runs portfolios of size i in iteration i, if there is a cost to CPU cycles, we achieve some savings relative to *Global* in iterations $i < k$. If the overhead for the evaluation of the portfolios after each iteration is bounded by ε, the CPU cycles used in *parHydra* are bounded by $\sum_{i=1}^{k} i \cdot (\frac{t_b}{k} + \varepsilon)$ as compared to $k \cdot (t_b + \varepsilon)$ for *Global*. If $k > 1$ and $\frac{t_b}{k} > \varepsilon$, *parHydra* will use fewer CPU cycles than *Global*.

Obviously, for $k > 1$, even if we assume that *AC* finds optimal configurations in each iteration, the *parHydra* procedure is not guaranteed to find a globally optimal portfolio. For instance, since the configuration found in the first iteration will be optimized to perform well on average on all instances I, the configuration added in the second iteration will then specialize to some subset of I. A combination of two configurations that are both specialized to different sets of instances may perform better; however, the configuration tasks in each *parHydra* iteration will be much easier than those performed by *Global* for even a moderately sized portfolio, giving us reason to hope that under realistic conditions, *parHydra* might perform better than *Global*, especially for large configuration spaces C and for comparatively modest time budgets t_b.

3.2.1.4 Independent configuration of component solvers (*Clustering*)

isac (Kadioglu et al., 2010; Malitsky & Sellmann, 2012) is a second method for automatically designing portfolio-based algorithm selectors. It works by clustering a set of instances in a given (normalized) instance feature space and then independently configuring the given highly parameterized algorithm on each instance cluster (see Algorithm 4). We adapted *isac* to the ACPP problem by making two generalizations. First, *isac* uses a linear normalization of the features, whereas we leave this decision as a parameter open to the user (for example, allowing standard, or so-called z-score, normalization). In general, the best normalization strategy is unknown and may vary between feature sets. Furthermore, there is no way to assess cluster quality before the configuration experiments are complete. Second, we set the number of

Algorithm 4: Portfolio Configuration Procedure *Clustering*

Input : parametric solvers with configuration space C; desired number k of component solvers; instance set I; performance metric m; configurator AC; number n of independent configurator runs; total configuration time t_b; feature normalizer FN; cluster algorithm CA; features $f(i)$ for all instances $i \in I$

Output: parallel portfolio solver with portfolio $\hat{c}_S$

1 normalize features with FN into feature space f'
2 cluster instances with CA in normalized feature space f' into k clusters S
3 **foreach** $s \in S$ **do**
4 **for** $j := 1..n$ **do**
5 obtain configuration $c_s^{(j)}$ by running AC with configuration space C on I_s using m for time $t_b/(k \cdot n)$, where I_s denotes all instances in cluster s
6 let $\hat{c}_s \in \arg\min_{c_s^{(j)} | j \in \{1...n\}} m(c_s^{(j)}, I)$ be the configuration which achieved best performance on I according to m
7 let $\hat{c}_S$ be the portfolio consisting the configurations for each clusters
8 **return** $\hat{c}_S$

clusters as a parameter, equaling the number of cores targeted by the parallel portfolio. Hence, we do not have to use a clustering method to determine how many clusters to choose (for example, *isac* uses g-means). To avoid suggesting that *isac*'s authors endorsed these changes, we refer to the resulting method using the neutral moniker *Clustering*. A key advantage of this approach is that execution of the configurator over clusters (Line 3) and over repetitions (Line 4) are independent and hence can be parallelized trivially, requiring overall wallclock time $t_b/(k \cdot n)$. However, *Clustering* performs the same amount of overall computation as *Global*, running k times n configuration experiments while *Global* runs n configuration experiments for a portfolio of size k; hence, *Clustering* is computationally more demanding than *parHydra*. A key disadvantage of the *Clustering* approach is that, unlike *Global* and *parHydra*, it requires instance features; moreover, these features should be suitable to induce homogeneous instance clusters in order to provide a good basis for automated configuration for those clusters (see also (Schneider & Hoos, 2012)).

3.2.2 Experiments

To empirically evaluate our methods for solving the ACPP problem, we applied *Global*, *parHydra* and *Clustering* to two state-of-the-art SAT solvers: *clasp* and *Lingeling*. Specifically, we compared our automatically configured parallel portfolios alongside performance-optimized sequential solvers, running on eight processor cores. Furthermore, we investigated the scalability of *parHydra* by assessing the performance of our portfolio after each iteration, thereby also assessing the slowdown observed for increasing number of component solvers due to hardware bottlenecks. Finally, we integrated our configured portfolio for *clasp* into *clasp*'s flexible multi-threading architecture and configured the clause sharing policy to investigate the influence of clause sharing on our trained ACPP solvers.

3.2.2.1 Scenarios

We compared six evaluation scenarios for each solver. We denote the default configuration of a single-process solver as *Default-SP* and that of a multi-process solver with 8 processes and without clause sharing as *Default-MP(8)*; *Default-MP(8)+CS* denotes the additional use of clause sharing, which is activated by default in both *Plingeling* and *clasp*. We contrasted these solver versions with four versions obtained using automated configuration: *Configured-SP* denotes the best (single-process) configuration obtained from configurator runs on a given training set, while *Global-MP(8)*, *parHydra-MP(8)* and *Clustering-MP(8)* represent the 8-component portfolios obtained using our *Global*, *parHydra* and *Clustering* methods. We chose this portfolio size to reflect widely available multi-core hardware, as used, for example, in the 2013 SAT Competition and also supported by the Amazon EC2 cloud (CC2 instances). However, our approach is not limited to eight cores but it scales as long as there are enough complementary configurations in the rich design space.

3.2.2.2 Solvers

We applied our approach to the SAT solvers *clasp* version 2.1.3 (Gebser et al., 2012c) and *Lingeling* version ala (Biere, 2012). We have selected *clasp* and *Lingeling* because they are state-of-the-art solvers for hardcombinatorial and industrial SAT instances since some time and therefore, the bar for demonstrating the efficacy of our ACPP approach is appropiately high. Furthermore, both solvers fulfil our only requirement for ACPP by being highly parameterized; *clasp* has 81 parameters and *Lingeling* has 118. Hence, the configuration space for 8 processes has 648 parameters for *clasp* and 944 parameters for *Lingeling*. We have not considered other state-of-the-art parameterized solvers, like *Glucose*, in these experiments, because *Glucose* has no parallelized counterpart for comparison with our automatically constructed solvers.

We did not apply our ACPP methods to *Plingeling*, the "official" parallel version of *Lingeling*, because it lacks configurable parameters for individual threads. We also disregarded the native parallel version of *clasp*, because *clasp*'s clause sharing mechanism, which cannot be turned off, results in highly non-deterministic runtime behaviour, rendering the configuration process much more difficult. We investigated the impact of clause sharing in a separate experiment. We executed all automatically constructed parallel portfolios via a simple wrapper script that runs a given number of solver instances independently in parallel and without communication between the component solvers.

3.2.2.3 Instance Sets

We conducted our experiments on instances from the *application* and *hard combinatorial* tracks of the 2012 SAT Challenge. Our configuration experiments made use of disjoint training and a test set, which we obtained by randomly splitting both instance sets into subsets with 300 instances each.[3]

[3] A random split into training and test set is often used in machine learning to get an unbiased performance estimate. However, such a simple split is pessimistic in its performance estimation. Because of the large amount of CPU resources needed for our experiments, we could not effort to measure the performance of our ACPP methods on more splits, for example, based on cross validation.

To ensure that our experiments would complete within a feasible amount of time, we made use of the instance selection technique proposed in (Hoos et al., 2013) on our training set to obtain a representative and effectively solvable subset of 100 instances for use with a runtime cutoff time of 180 seconds. As a reference for the selection process, we used the base features of *SATzilla* (Xu et al., 2008) and *SINN* (Yasumoto, 2012), *Lingeling* (Biere, 2012), *Glucose* (Audemard & Simon, 2012), *clasp* (Gebser et al., 2012c) and *CCASat* (Cai et al., 2012) as representative set of state-of-the-art solvers, as also proposed in (Hoos et al., 2013).

3.2.2.4 Resource Limits and Hardware

We chose a cutoff time of 180 seconds for algorithm configuration on the training set and 900 seconds for evaluating solvers on the test set (as in the 2012 SAT Challenge). Additionally, we performed three repetitions of each solver and test instance run and report the median of those three runs. All solver runs (on both training and test sets) were restricted to use at most 12 GB of memory (as in the 2012 SAT Challenge). If a solver was terminated because of memory limitations, we recorded it as a timeout. We performed all solver and configurator runs on Dell PowerEdge R610 systems with two Intel Xeon E5520 CPUs with four cores (2.26GHz) running 64-bit Scientific Linux (2.6.18-348.6.1.el5).

3.2.2.5 Configuration Experiments

We performed configuration using *SMAC* (version 2.04.01) (Hutter et al., 2011a), a state-of-the-art algorithm configurator. *SMAC* allows the user to specify the initial incumbent, as required in the context of our *parHydra* approach (see Lines 2 and 5 of Algorithm 3). We specified PAR10 as our performance metric, and gave *SMAC* access to the base features of *SATzilla* (Xu et al., 2008). (*SMAC* builds performance models internally; it can operate without instance features, but often performs better when they are available.) To enable fair performance comparisons, in the case of *Configured-SP* ($n = 80$) and *Global-MP(8)* ($n = 10$) we allowed 80 hours of configuration time and 2 hours of validation time, which amounts to a total of 6560 CPU hours for $k = 8$. For *parHydra-MP(8)*, we allowed for 10 hours of configuration time and 2 hours of validation time per configurator run ($n = 10$) in each iteration, which amounts to a total of 3360 CPU hours. When using a cluster of dedicated machines with 8-core CPUs, each of these solver versions could be produced within 96 hours of wall-clock time. For *Clustering-MP(8)* ($n = 10$), we allowed for 10 hours of configuration time and 2 hours of validation time, which also amounts to a total of 6560 CPU hours for $k = 8$. On a cluster, a parallel solver with this approach could be produced within 12 hours of wall-clock time. Even though, we used a large amount of CPU resources, the ACPP process is fully automatic so that no human intervention is needed. Therefore, more valuable human time is saved by avoiding the need of implementing a parallel solver from scratch.

Clustering-MP(8) used k-means with random initial centroids, 1000 restarts and the base features of *SATzilla* (Xu et al., 2008), the same features as used by *isac* (Kadioglu et al., 2010). Since the right choice of the feature normalization strategy can vary between applications, we considered three standard methods from literature, namely, no feature normalization, denoted as *Clustering-None-MP(8)*, linear min-max feature normalization to a range of $[-1, 1]$ (as used by *isac*), denoted as *Clustering-Linear-MP(8)*, and z-score feature normalization (mean 0 and

	Lingeling (*application*)			*clasp* (*hard combinatorial*)		
Solver Set	#TOs	PAR10	PAR1	#TOs	PAR10	PAR1
Default-SP	72	2317	373	137	4180	481
Configured-SP	68	2204	368	140	4253	473
Global-MP(8)	**52***	**1702***	**298***	98	3011	365
parHydra-MP(8)	55*†	1788*†	**303***†	**96***†	**2945***†	**353***†
Clustering-None-MP(8)	**47***	**1571***	**302***	107	3257	368
Clustering-Linear-MP(8)	61	1970	323	114	3476	398
Clustering-Zscore-MP(8)	**51***	**1674***	**297***	99	3035	362
Default-MP(8)	64	2073	345	96	2950	358
Default-MP(8)+CS	**53***	**1730***	**299***	**90***	**2763***	**333***

Table 3.1: Runtime statistics on the test set from *application* and *hard combinatorial* SAT instances achieved by single-processor (SP) and 8-processor (MP8) versions. *Default-MP(8)* was *Plingeling* in case of *Lingeling* and `clasp -t 8` for *clasp* where both use clause sharing (CS). The performance of a solver is shown in boldface if it was not significantly different from the best performance, and is marked with an asterisk (*) if it was not significantly worse than *Default-MP(8)+CS* (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$). The best ACPP portfolio on the training set was marked with a dagger (†).

standard deviation 1; assuming some normal distribution properties of the features), denoted as *Clustering-Zscore-MP(8)*.

3.2.2.6 Results and Interpretation

To evaluate our ACPP solvers, we present the number of timeouts (#TOs), PAR10 and PAR1 based on the median performance of the three repeated runs for each solver–test instance pair in Table 3.1. The best ACPP portfolio on the training set was marked with a dagger (†) to indicate that we would have chosen this portfolio if we had to make a choice only on the trainings data. Furthermore, we applied a statistical test (a permutation test with 100 000 permutations and significance level $\alpha = 0.05$) to the (0/1) timeout scores, the PAR10 scores and the PAR1 scores to determine whether performance differences of the solvers were significant. In Table 3.1, performance of a given solver is indicated in bold face if it was not significantly different from the performance of the best solver. Furthermore, we use an asterisk (*) to indicate that a given solver's performance was not significantly worse than the performance of *Default-MP(8)+CS*—the official parallel solver with clause sharing produced by experts.

Table 3.1 summarizes the results of experiments with *Lingeling* and *clasp*. Running a configurator to obtain an improved, single-processor solver (*Configured-SP*) made a statistically insignificant impact on performance. We thus believe that these default configurations are nearly optimal, reflecting the status of *Lingeling* and *clasp* as state-of-the-art solvers. With *Lingeling* as the component solver, *Clustering-None-MP(8)* produced the best-performing portfolio. The portfolio of *Clustering-None-MP(8)* also significantly outperformed *parHydra-MP(8)* on time-

out scores and PAR10 scores, but not on PAR1 scores. There was no significant difference on any of these scores between *Clustering-None-MP(8)*, *Clustering-Zscore-MP(8)*, *Global-MP(8)* and *Default-MP(8)+CS* and also no significant difference between *parHydra-MP(8)* and *Default-MP(8)+CS*. However, the portfolio performance of both *Clustering-Linear-MP(8)* and *Default-MP(8)* (*Plingeling* with deactivated clause sharing) was significantly worse than the performance of all other parallel portfolios and not even significantly better than *Configured-SP* in terms of timeout scores or PAR10 scores. Note that *Plingeling* (without clause sharing) builds a parallel portfolio only in a degenerate sense, simply using different random seeds and thus making different choices in the default phase (Biere, 2012). Hence, it is not surprising that *Plingeling* without clause sharing performed significantly worse than *Plingeling* with clause sharing.

With *clasp* as the component solver, the portfolio constructed by *parHydra-MP(8)* was the best ACPP solver and matched (up to statistically insignificant differences) the performance of *Default-MP(8)+CS* (the expert-constructed portfolio solver with clause sharing) according to all metrics we considered, despite incurring six more timeouts. All other ACPP solvers fell short of this (high) bar; however, the portfolios of *Global-MP(8)* and *Clustering-Zscore-MP(8)* performed as well as the default portfolio of *clasp* without clause sharing (*Default-MP(8)*). While *Clustering-None-MP(8)*'s portfolio and *Clustering-Linear-MP(8)*'s portfolio performed significantly worse than *Default-MP(8)*, all parallel solvers significantly outperformed the single-threaded versions of clasp.

We note that *Clustering-MP(8)* clusters the training instances based on instance features; thus, normalizing these features in different ways can result in different instance clusters. There is no way to assess cluster quality before configuration experiments are complete; one can only observe the distribution of the instances in the clusters. For example, the instances in the training set of the *application* distribution for *Clustering-None-MP(8)* were distributed across clusters with 2, 2, 3, 11, 13, 18, 21, and 30 instances per cluster; we observed qualitatively similar distributions for *Clustering-Linear-MP(8)* and *Clustering-Zscore-MP(8)*. This is potentially problematic, because running a configurator on sets of 2 or 3 instances can lead to overfitting and produce configurations whose performance does not generalize well to new instances. In (Kadioglu et al., 2010), Kadioglu et al. described how *isac* removes such small clusters by integrating them in larger clusters. However, the number of clusters is fixed in the case of parallel portfolios because the number of clusters has to match the size of the portfolio to use the parallel resources to their fullest.

For both solvers, linear feature normalization (*Clustering-Linear-MP(8)*) produced clusters that were insufficiently complementary, and hence led to relatively poor performance. (We note that linear normalization is used in *isac*.) Using clustering without feature normalization (*Clustering-None-MP(8)*) led to surprisingly strong performance in the case of *Lingeling* on the *application* instances, but failed to reach the performance of *Default-MP(8)+CS* for *clasp* on the *hard combinatorial* scenario. Similarly, the use of z-score normalization (*Clustering-Zscore-MP(8)*) did not produce portfolios that consistently reached the performance of *Default-MP(8)+CS*.

Finally, *parHydra-MP(8)* was the only ACPP solver that matched the performance of *Default-MP(8)+CS* on both domains. *parHydra-MP(8)*'s portfolio had also the best training performance and therefore, out of the ACPP solvers, we would choose it. However, while *Default-MP(8)+CS* uses clause sharing, *parHydra-MP(8)* does not. This is surprising, because the performance of *Plingeling* and *clasp* without clause sharing was significantly worse than with clause sharing. Thus, *parHydra-MP(8)* was the best performing method among those that did not perform clause

	Lingeling (*application*)			*clasp* (*hard combinatorial*)		
Solver	#TOs	PAR10	PAR1	#TOs	PAR10	PAR1
Default-SP	72	2317	373	137	4180	481
parHydra-MP(1)	82	2594	380	136	4136	464
parHydra-MP(2)	65	2086	331	118	3607	421
parHydra-MP(3)	60	**1933**	313	115	3515	410
parHydra-MP(4)	**56**	**1874**	**308**	115	3507	402
parHydra-MP(5)	**58**	**1878**	312	105	3219	384
parHydra-MP(6)	60	1935	315	103	3161	380
parHydra-MP(7)	59	1902	309	102	3126	372
parHydra-MP(8)	***55***	**1788**	**303**	**96**	**2945**	**353**

Table 3.2: Runtime statistics of *parHydra-MP(i)* after each iteration *i* (test set). The performance of a solver is shown in boldface if it was not significantly different from the best performance, (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$).

sharing.

3.2.2.7 Scalability and Overhead

Although 8-core machines have become fairly common, 4-core machines are still more commonly used as desktop computers. Furthermore, in (Asin, Olate, & Ferres, 2013), Asin et al. observed that parallel portfolios scale sublinearly in the number of cores—in part, because component solvers share the same CPU cache. Therefore, we investigated how the performance of our automatically constructed portfolio scales with the number of processors. The *parHydra* approach has the advantage that the portfolio is extended by one configuration at each iteration, making it easy to perform such scaling analysis.

Table 3.2 shows the test-set performance of *parHydra-MP(i)* after each iteration. First of all, *parHydra-MP(1)* was able to find a better performing configuration than *Default-SP* for *clasp*. In contrast, *parHydra-MP(1)* found a poorly performing configuration for *Lingeling* in comparison to *Default-SP*, and had to compensate in subsequent iterations. For both solvers, the largest performance improvement occurred between the first and second iterations, with the number of timeouts reduced by 17 for *Lingeling* and 18 for *clasp*. In later iterations, performance can stagnate or even drop: for example, *parHydra-MP(5)* solves two more instances than *parHydra-MP(6)* with *Lingeling*. This may in part reflect hardware limitations: as the size of a portfolio increases, more processes compete for fixed memory (particularly, cache) resources.

We investigated the influence of these hardware limitations on the performance of our parallel solvers by constructing portfolios consisting of identical copies of the same solver. In particular, we replicated the same configuration multiple times with the same random seed; clearly, this setup should result in worsening performance as portfolio size increases, because each component solver does exactly the same work but shares hardware resources. (We note that these experiments are particularly sensitive to the underlying hardware we used.) To compare directly against Table 3.2, we used the configurations found in the first iteration of *parHydra-MP(1)*. In

	Lingeling (*application*)			*clasp* (*hard combinatorial*)		
# Processes	#TOs	PAR10	PAR1	#TOs	PAR10	PAR1
1	82	2594	380	136	4136	464
2	**79**	**2509**	**376**	**134**	**4079**	**461**
3	85	2509	376	135	4106	451
4	86	2677	382	135	4107	452
5	89	2707	385	135	4108	463
6	90	2793	390	135	4110	465
7	90	2820	390	135	4110	465
8	92	2877	393	136	4139	467

Table 3.3: Runtime statistics of *Lingeling* and *clasp* with parallel runs of the same configuration on all instances in the corresponding test sets. The performance of a solver is shown in boldface if it was not significantly different from the best performance, (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$).

Table 3.3, we see that hardware limitations do seem to impact the portfolio of *Lingeling* solvers; for example, a single *Lingeling* configuration solves 10 more instances than eight such configurations running in parallel on an eight-core machine. In contrast, the performance of *clasp* varied only slightly as duplicate solvers were added. Based on the results in (Aigner, Biere, Kirsch, Niemetz, & Preiner, 2013), we suspected that this overhead arose because of memory issues, noting that we evaluated *clasp* on *hard combinatorial* instances with an average size of 1.4 MB each, whereas we evaluated *Lingeling* on *application* instances with an average size of 36.7 MB. We confirmed that *clasp*'s portfolio did experience overhead on instances with large memory consumption, and that *Lingeling* produced nearly no overhead on instances with low memory consumption.

An interesting further observation is that *Lingeling* and *clasp* performed best if two copies of the same configuration ran in parallel and that running only one copy was worse than two copies. We can only speculate about the reasons which may are connected with cache misses or something similar.

3.2.2.8 Algorithm Configuration of Clause Sharing

Our previous experiments did not allow our component solvers to share clauses, despite evidence from the literature that this can be very helpful (Hamadi et al., 2009b). The implementation of clause sharing is a challenging task; for example, if too many clauses are shared, the overhead caused by clause sharing may exceed the benefits (Lazaar, Hamadi, Jabbour, & Sebag, 2012). Furthermore, the best clause sharing policy varies across instance sets. In the following, we investigate the application of clause sharing on our ACPP portfolio. Since there are a lot of possible clause sharing policies, we again use algorithm configuration for the purpose of identifying effective clause sharing policies. This can be understood as an additional instrument to improve the performance of ACPP portfolios if clause sharing is already available.

To study the impact of clause sharing on our ACPP procedures, we relied upon the clause sharing infrastructure provided by *clasp* (Gebser et al., 2012c), which has a relatively highly

clasp variant	#TOs	PAR10	PAR1
Default-MP(8)	96	2950	358
Default-MP(8)+CS	**90**	**2763**	**333**
parHydra-MP(8)	96	2945	**353**
parHydra-MP(8)+defCS	**90**	**2777**	**347**
parHydra-MP(8)+confCS	**88**	**2722**	**346**

Table 3.4: Runtime statistics of *clasp*'s *parHydra-MP(8)* portfolio with default clause sharing (defCS) and configured clause sharing (confCS) on the test instances of the *hard combinatorial* set. The performance of a solver is shown in boldface if its performance was at least as good as that of any other solver, up to statistically insignificant differences (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$).

parametrized clause sharing policy (10 parameters) and allows for the configuration of each component solver. *Plingeling*, on the other hand, does not support the configuration of each component solver. As before, we considered the *hard combinatorial* instance set.

We started with the portfolio identified by *parHydra-MP(8)*. *clasp*'s multi-threading architecture performs preprocessing before threading is used. Hence, we ignored the preprocessing parameters identified in the *parHydra-MP(8)* portfolio, adding them again to the configuration space as global parameters. Since the communication of clause sharing induces greater variation in solving behaviour, we used 50 CPU hours as the configurator's time budget.

Table 3.4 shows the performance of *clasp*'s default portfolio with clause sharing, *Default-MP(8)+CS*; the portfolio originally returned by *parHydra*, which does not perform clause sharing, *parHydra-MP(8)*; the application of *clasp*'s default clause sharing and preprocessing settings to the original *parHydra* portfolio, *parHydra-MP(8)*+defCS; and the *parHydra* portfolio with newly configured clause sharing and preprocessing settings, *parHydra-MP(8)*+confCS. As confirmed by these results, the use of clause sharing led to significant performance gains; furthermore, while the additional gains through configuring the clause sharing and preprocessing mechanisms were too small to reach statistical significance, *parHydra-MP(8)*+confCS solved two more instances than *Default-MP(8)+CS* and *parHydra-MP(8)*+defCS.

We note that there is potential for performance to be improved even further if clause sharing were configured alongside the portfolio itself. For example, *clasp*'s default portfolio contains configurations that are unlikely to solve instances directly, but that generate useful clauses for other clasp instances.[4] Clearly, our methodology for configuring clause sharing will not identify such configurations. Configuration of clause sharing can be directly integrated in *Global* and *parHydra* because the solvers are actually running in parallel. However, since the solver with clause sharing is highly non-deterministic, the configuration process should get a lot more time to construct the portfolio. Related to this, some results in the literature indicate that the collaboration of SAT solvers via clause sharing is more natural if the solvers uses similar strategies, for example, the same solver with a fixed configuration runs several times in parallel but with different seed (e.g., *Plingeling*). If the configuration of the portfolio is done alongside the configuration of the clause sharing policy, such homogeneous portfolios would be also in

[4]Personal communication with the main developer of *clasp*, Benjamin Kaufmann.

the design space of our ACPP methods. We plan to investigate other approaches in future work.

3.2.2.9 Conclusion

Given a solver with a rich design space (such as *Lingeling* and *clasp*), all our ACPP methods were able to generate parallel solvers with 8 cores that significantly outperform their sequential counterparts - although 2 cores were already enough to do so. Therefore, we were able to show that our ACPP methods are able to automatically build parallel portfolio solvers without the need to start from scratch to get an efficient parallel SAT solver. However, the analysis of the scalability showed that hardware restrictions incur overhead if more processor cores are used. The scalability of our ACPP methods is therefore limited by the richness and complementarity of the solver's design space. Furthermore, we were able to verify that clause sharing can be used to improve the performance of our ACPP solver even more and should be also adjusted with algorithm configuration. Nevertheless, we note that our ACPP methods do not depend on the availability of clause sharing to generate efficient parallel solvers.

3.3 Parallel Portfolio Configuration with Multiple Sequential Solvers

So far, we have shown that our procedures are able to construct effective parallel portfolios based on single solvers with rich design spaces. There is considerable evidence from the literature and from SAT competitions that strong portfolios can also be built by combining entirely different solvers in their default configurations (see, for example, SATzilla (Xu et al., 2008), *ppfolio* (Roussel, 2011) and *pfolioUZK* (Wotzlaw et al., 2012)). For instance, *ppfolio* was simply build by taking the best solver from the last competition and combining them in a portfolio. *pfolioUZK* considered some more state-of-the-art solvers and made some simple experiments to find the best combination of solvers in a portfolio. Both portfolios do not consider the configuration space of the component solvers and therefore, they are simple baselines for our ACPP approach. However, *ppfolio* and *pfolioUZK* use *Plingeling* as a portfolio component. Since we want to investigate the strength of our ACPP methods without the human expert knowledge on parallel solving, we consider first only sequential solvers to construct ACPP solvers. This section and the following section investigates the extension of our automatic techniques to the construction of portfolios based on the configuration spaces spanned by such solver sets.

3.3.1 Approach

As long as all of our component solvers are sequential, we can simply use the ACPP procedures defined in Section 3.2. We can accommodate the multi-solver setting by introducing a solver choice parameter for each portfolio component (see Figure 3.1). The parameters of solver $a \in A$ are only active when the solver choice parameter is set to use a. This is implemented by using conditional parameters (see the PCS format of the Algorithm Configuration Library (Hutter, Lopez-Ibanez, Fawcett, Lindauer, Hoos, Leyton-Brown, & Stützle, 2014a)). Similar architectures were used by *SATenstein* (KhudaBukhsh, Xu, Hutter, Hoos, & Leyton-Brown, 2009) and *Auto-WEKA* (Thornton et al., 2013).

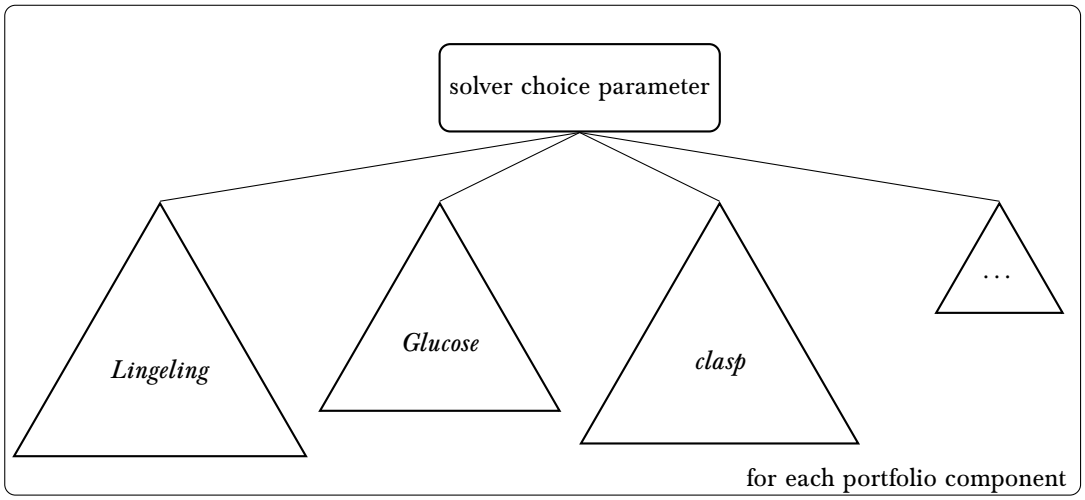


Figure 3.1: Using a solver choice parameter, we can specify a single configuration space that spans multiple solvers.

We have so far aimed to create portfolios with size equal to the number of available processor cores. But as observed in Section 3.2.2.7, each component solver used within a parallel portfolio incurs some overhead. A similar observation was made by the developer of *pfolioUZK* (personal communication) and prompted the decision for *pfolioUZK* to use only 7 components on an 8-core platform. To allow our portfolios to make the same choice, we included "none" as one of choices available for each portfolio component.

3.3.2 Experiments

While we would presumably have obtained the strongest parallel solver by allowing our portfolio to include a very wide range of modern SAT solvers, this would have made it difficult to answer the question how our automated methods compare to human expertise in terms of the performance of the parallel portfolios thus obtained. In particular, we were interested in *pfolioUZK* (Wotzlaw et al., 2012), a state-of-the-art parallel solver that won the parallel track of the 2012 SAT Challenge with *application* instances. To compare our automatic methods with the manual efforts of *pfolioUZK*'s authors, we thus chose the same set of solvers they considered as the basis for our experiments. This allows us to fairly assess the strength of our automated portfolio generation methods.

3.3.2.1 Solvers

pfolioUZK uses *satUZK*, *Lingeling*, *tnm*, and *MPhaseSAT_M* on the same core in its sequential version (*Default-SP*) and *satUZK*, *Glucose*, *contrasat* and *Plingeling* with 4 threads and clause sharing in its 8-process parallel version (*Default-MP(8)+CS*). In all cases, solvers are used in their default configurations. However, in designing *pfolioUZK* (Wotzlaw et al., 2012), Wotzlaw et al. considered the following, larger set of component solvers:

- *contrasat* (van Gelder, 2012): 15 parameters

8-Processor Parallel Solver	#TOs	PAR10	PAR1
pfolioUZK-ST	150	4656	606
pfolioUZK-MP(8)+CS	**35**	**1168**	**223**
Global-MP(8)(*pfolioUZK* w/o *Plingeling*)	44	1463	275
parHydra-MP(8)(*pfolioUZK* w/o *Plingeling*)	**39**†	**1297**†	244†
Clustering-None-MP(8)(*pfolioUZK* w/o *Plingeling*)	42	1390	256
Clustering-Linear-MP(8)(*pfolioUZK* w/o *Plingeling*)	48	1581	285
Clustering-Zscore-MP(8)(*pfolioUZK* w/o *Plingeling*)	52	1676	272

Table 3.5: Runtime statistics for 8-processor parallel solvers on the *application* test set. The performance of a solver is shown in boldface if it was not significantly different from the best performance (according to a permutation test with 100 000 permutations at significance level $\alpha = 0.05$). The best ACPP portfolio on the training set was marked with a dagger (†).

- *Glucose* 2.0 (Audemard & Simon, 2012): 10 parameters for *satelite* preprocessing and 6 for *Glucose*
- *Lingeling* 587 (Biere, 2011): 117 parameters
- *march_hi* 2009 (Heule et al., 2004): 0 parameters
- *MPhaseSAT_M* (Chen, 2011): 0 parameters
- *satUZK* (Grinten, Wotzlaw, Speckenmeyer, & Porschen, 2012): 1 parameter
- *sparrow2011* (Tompkins, Balint, & Hoos, 2011): 0 parameters[5]
- *tnm* (Li, Wei, & Li, 2012): 0 parameters

Overall, the configuration space we considered has 144 parameters for each portfolio component, and thus 1152 parameters for an 8-component parallel portfolio.

3.3.2.2 Instances and Setup

We evaluated *pfolioUZK* and our *Global*, *parHydra*, and *Clustering* approaches on the same 300 *application* test instances of the 2012 SAT Challenge as used before. Otherwise, our experimental setup was as described in Section 3.2.2.

3.3.2.3 Results and Interpretation

The first part of Table 3.5 shows the results of *pfolioUZK* in its sequential and parallel versions. Recall that *pfolioUZK* uses *Plingeling* with clause sharing as a component solver. Sequential *pfolioUZK* experienced 114 more timeouts than its parallel version; indeed, it was only ranked 16th in the sequential application track of the 2012 SAT Challenge.

The second part of Table 3.5 summarizes the performance of our ACPP solvers (which do not use *Plingeling* as a component solver). *parHydra-MP(8)* performed best; indeed, there

[5] Although *sparrow2011* should be parameterized (Tompkins et al., 2011), the source code and binary provided with *pfolioUZK* does not expose any parameters.

was no significant difference between *parHydra-MP(8)* and *pfolioUZK*-MP(8) in terms of timeout and PAR10 scores. This indicates that our ACPP approach is indeed able to match the performance of parallel portfolios manually constructed by experts, even with the disadvantage of being prohibited from using *Plingeling* and thus clause sharing. *Global-MP(8)* and *Clustering-None-MP(8)* performed significantly worse than *pfolioUZK*-MP(8), but not significantly worse than *parHydra-MP(8)* in terms of timeout and PAR10 scores.

As we previously observed with portfolios based on *Lingeling*, *Clustering-None-MP(8)* (no feature normalization) performed best among the *Clustering* approaches. However, this time, *Clustering-Zscore-MP(8)* performed worse than *Clustering-Linear-MP(8)*. This indicates that the quality of the clusters depends not only on the instance set but also on the configuration space of the portfolio (which, indeed, is disregarded by the *Clustering* approach).

Although we allowed our portfolio-building procedures to choose "none" for any component solver, this option was never selected. We note that the component solvers of all *Clustering* approaches are configured independently; "none" would thus never be chosen by any *Clustering* approach.

3.3.2.4 Conclusion

The use of a set of complementary solvers and exploiting their configuration space lead to even better performing ACPP solvers in comparison to using only one solver such as *Lingeling*(compare Table 3.1 and Table 3.5). To get such an ACPP solver, we did not need to modify our ACPP methods but we used conditionals in our configuration space to distinguish between the design spaces of the individual solvers. However, since we did not used parallel solvers with clause sharing, such as *Plingeling*, in our portfolio, our *parHydra* method was able to generate a parallel solver without clause sharing as good as *pfolioUZK* but was not able to outperform it.

3.4 Parallel Portfolio Configuration with Multiple Sequential and Parallel Solvers

Our results reported so far confirm the intuition that clause sharing is an important ingredient of high-performance parallel solvers. This section extends the scope of our ACPP methods to allow inclusion of parallel solvers that perform clause sharing as portfolio components. By this, we combine our automatic methods with the human expert knowledge to boost the solving performance even further.

3.4.1 Approach

To add parallel solvers as portfolio components, we consider them as single solvers with large configuration spaces rather than multiple copies of solvers with smaller configuration spaces. This allows us to set parameters of parallel solvers that are common to several threads or processes, such as those that control clause sharing.

3.4.1.1 *parHydra*$_b$

The components of *Plingeling* are not parameterized. If the portfolio can also consist of configured versions of *Lingeling*, which subsumes single-core *Plingeling*, and the configurator is run for long enough, there is no reason for the *parHydra* approach to choose *Plingeling* as a component, unless *Plingeling* already belongs to the previous iteration's portfolio (in which case the benefits of clause sharing can make themselves felt). Obviously then, an argument by induction shows that *Plingeling* will *never* be added by *parHydra*, revealing a disadvantage of its greedy optimization strategy. *Global* does not have this problem, but has difficulties dealing with the large configuration space encountered here.

To overcome both of these limitations and effectively interpolate between *parHydra* and *Global*, we introduce a new approach, which we call *parHydra*$_b$ (Algorithm 5). In brief, unlike *parHydra*, *parHydra*$_b$ simultaneously configures b processes in each iteration. Specifically, in Lines 2 and 3, *parHydra*$_b$ iterates up to the desired number of component solvers with a step size of b; in Line 5, the algorithm configurator is used to find a portfolio of b configurations with b times the configuration time budget and adds them to the current portfolio $c^{(j)}_{1:i'}$. After the n independent runs of the algorithm configurator (Line 4 and 5), the best performing portfolio $\hat{c}_{1:i'}$ is selected in Line 6, and in Line 7, the initial incumbent for the next iteration is selected based on the marginal contribution to the currently selected portfolio. The parameter b controls the size of the configuration space in each iteration. Since the configuration space grows exponentially with b but we allow configuration time to grow only linearly, the algorithm configurator has a harder task under *parHydra*$_b$ than under *parHydra*. However, for sufficiently small b, this additional cost can be worthwhile, because of *parHydra*$_b$'s reduced tendency to stagnate in local minima.

3.4.1.2 Clustering

The *Clustering* approach cannot be effectively applied to sets of component solvers that include parallel solvers. When the configuration of each component solver is performed independently of all other solvers, there is no way to direct a configurator to consider synergies between solvers, such as those arising from clause sharing. Therefore, an unparameterized, parallel solver with clause sharing, such as *Plingeling*, will never be selected. Thus, we did not consider a variant of *Clustering* in the experiments described below.

3.4.2 Experiments

We used the set of solvers described in Section 3.3.2, with the addition of *Plingeling*. We added *parHydra*$_b$ to the set of ACPP methods considered and allowed $b \in \{2,4\}$. We use the same setup as before, except that we allowed a 20-hour configuration budget per configured process, twice as much as before, to take into consideration the greater variation in solving behaviour of *Plingeling* which induces a harder configuration task.

We compared our results to a variety of state-of-the-art solvers on this benchmark set. We considered two state-of-the-art sequential solvers: *Glucose* (2.1) (Audemard & Simon, 2012) (winner of the single-engine application track—like all other competition results cited below, in the 2012 SAT Challenge); and *SATzilla-App* (Xu, Hutter, Shen, Hoos, & Leyton-Brown, 2012b),

Algorithm 5: Portfolio Configuration Procedure *parHydra*$_b$

Input : set of parametric solvers $a \in A$ with configuration space C_a; desired number k of component solvers; number b of component solvers simultaneously configured per iteration; instance set I; performance metric m; configurator AC; number n of independent configurator runs; total configuration time t_b

Output: parallel portfolio solver with portfolio $\hat{c}_{1:k}$

$i := 1$
while $i < k$ **do**
  $i' := i + b - 1$
  for $j := 1..n$ **do**
    obtain portfolio $c^{(j)}_{1:i'} := \hat{c}_{1:i-1} || c^{(j)}_{i:i'}$ by running AC on configuration space $\{\hat{c}_{1:i-1}\} \times (\prod_{l=i}^{i'} \bigcup_{a \in A} \{(c) \mid c \in C_a\})$ and initial incumbent $\hat{c}_{1:i-1} || c_{init}$ on I using m for time $t_b \cdot b/(k \cdot n)$
  let $\hat{c}_{1:i'} \in \arg\min_{c^{(j)}_{1:i'} | j \in \{1...n\}} m(c^{(j)}_{1:i'}, I)$ be the configuration that achieved best performance on I according to m
  let $c_{init} \in \arg\min_{c^{(j)}_{i:i'} | j \in \{1...n\}} m(\hat{c}_{1:i'} || c^{(j)}_{i:i'}, I)$ be the configuration that has the largest marginal contribution to $\hat{c}_{1:i'}$
  $i := i + b$
return $\hat{c}_{1:k}$

which is *SATzilla* trained on application instances (winner of the sequential portfolio application track). We also considered the following high-performance parallel solvers[6]:

- *clasp* (2.1.3) (Gebser et al., 2012c);
- *Plingeling* (ala) (Biere, 2012) and *Plingeling*(aqw) (Biere, 2013)[7];
- *ppfolio* (Roussel, 2011) (bronze medal in the parallel track);
- *PeneLoPe* (Audemard et al., 2012) (silver medal in the parallel track);
- and again *pfolioUZK* (Wotzlaw et al., 2012) (winner of the parallel track).

The first part of Table 3.6 summarizes the performance results for these solvers: first the sequential solvers in their default configurations (*Default-SP*), then the parallel solvers using clause sharing in their default configurations (*Default-MP(8)+CS*), and finally our ACPP solvers based on the component solvers of *pfolioUZK*. As already discussed, the performance of the sequential *pfolioUZK* does not achieve the state-of-the-art performance; this distinction goes to

[6] We did not considered *3Spar* and *CSCHpar*, parallel algorithm selection solvers, here because the only availabe versions are optimized for a mixed set of SAT instances (application, handcrafted and random) and there is no trainable version available. Therefore, a fair comparison between them and our ACPP portfolios is not be possible.

[7] The process of implementing, benchmarking and writing the paper took more than one year; so that a new SAT Competition (2013) took place and the new *Plingeling* aqw version won the gold medal in the parallel track.

Solver	#TOs	PAR10	PAR1
Single threaded solvers: *Default-SP*			
pfolioUZK-ST	150	4656	606
Glucose-2.1	55	1778	293
SATzilla-2012-*APP*	38	1289	263
Parallel solvers with default config: *Default-MP(8)*			
Plingeling(ala)+CS	53	1730	299
PeneLoPe+CS	49	1563	240
ppfolio+CS	46	1506	264
clasp+CS	37	**1203**	**204**
pfolioUZK-MP8+CS	35	1168	223
Plingeling(aqw)+CS	**32**	**1058**	**194**
ACPP solvers including a parallel solver			
parHydra-MP(8)(*pfolioUZK*)	34	1143	225
parHydra$_2$*-MP(8)*(*pfolioUZK*)	32	1082	218
parHydra$_4$*-MP(8)*(*pfolioUZK*)	**29**†	**992**†	**209**†
Global-MP(8)(*pfolioUZK*)	35	1172	227

Table 3.6: Comparison of parallel solvers with 8 processors on the test set of *application*. The performance of a solver is shown in boldface if its performance was at least as good as that of any other solver, up to statistically insignificant differences (according to a permutation test with 100 000 permutations at significance level $\alpha = 0.05$). The best ACPP portfolio on the training set was marked with a dagger (†).

Glucose for a single solver, and *SATzilla* for a portfolio-based algorithm selector. The performance differences between all three of these solvers were statistically significant.

pfolioUZK and *clasp* performed significantly better than *ppfolio*, *PeneLoPe* and *Plingeling*; we observed no significant performance difference between *pfolioUZK* and *clasp* in terms of any of the scores we measured. (Even with further, extensive experiments, we have not been able to determine why *clasp* performed significantly worse than *pfolioUZK* and *Lingeling* in the 2012 SAT Challenge.)

parHydra$_4$*-MP(8)* produced the best parallel portfolio solver overall, which turned out to be significantly faster than *pfolioUZK*. The portfolio solvers produced by *parHydra-MP(8)* and *parHydra*$_2$*-MP(8)* exhibited no significant performance differences from *pfolioUZK*. Furthermore, *parHydra*$_4$*-MP(8)* also solved more instances than *Plingeling*(aqw) although *Plingeling*(aqw) won the 2013 SAT competition and the solvers in *parHydra*$_4$*-MP(8)* were mostly published in 2011, which gives *Plingeling*(aqw) two more years of development.

Taking a closer look at these portfolio solvers, *parHydra*$_2$*-MP(8)*, *parHydra*$_4$*-MP(8)* and *Global-MP(8)* allocated three cores to *Plingeling*. As expected, *parHydra-MP(8)* did not include *Plingeling* in its portfolio; however, it did include three variants of *Lingeling*. All four portfolio solvers used at most seven processes by selecting "none" on one process; *Global-MP(8)* selected "none" twice.

3.4.2.1 Conclusion

Using our extended *parHydra*$_b$ method and a parallel solver with clause sharing, our *parHydra*$_b$ was able to generate an ACPP solver outperforming *pfolioUZK* and being at eye level with *Plingeling* (aqw) which used a lot more modern solving strategies than used in baseline portfolio from *pfolioUZK*. This shows that the combination of our automatic ACPP methods in combination with the knowledge of an expert can be not only used to generate efficient parallel solvers but also to improve the state-of-the-art in parallel SAT solving. So our ACPP method can also used to support an expert in parallel solving to build parallel solvers.

3.5 Conclusion

In this work, we demonstrated that sequential algorithms can be combined automatically and effectively into parallel portfolios, following an approach we call Automatic Construction of Parallel Portfolios (ACPP). This approach enables solver developers to leverage parallel resources without having to be concerned with synchronization, race conditions or other difficulties that arise in the explicit design of parallel code. However, we acknowledge that parallel solving techniques like clause sharing can further improve the performance of our ACPP portfolios.

We investigated three different ACPP procedures: (i) configuration in the joint configuration space of all portfolio components (*Global*); (ii) configuration on a set of instance clusters (*Clustering*); and (iii) iteratively adding one or more component solvers at a time (*parHydra*). We assessed these procedures on widely studied classes of satisfiability problems: the *application* and *hard combinatorial* tracks of the 2012 SAT Challenge. Overall, we found that *parHydra* was the most practical method. The configuration space of *Global* grows exponentially with the size of the portfolio; thus, while in principle it subsumes the other methods, in practice, it tended not to find state-of-the-art portfolios within available time budgets. *Clustering* also tended not to yield state-of-the-art portfolios; furthermore, unlike our other methods, *Clustering* relies on a set of instance features, and is hence sensitive to feature normalization. We experimented with different approaches, and found that the best approach varied from one setting to another. In contrast to *Global* and *Clustering*, *parHydra* was able to find state-of-the-art portfolios on all of our domains, even improved the state-of-the-art on *application* instances using *pfolioUZK*'s solvers and was able to keep up with the state-of-the-art from one year later, that is, *Plingeling*(aqw) won the 2013 parallel track. We expect that as additional highly parametric SAT solvers become available, *parHydra* will produce even stronger parallel portfolios.

In future work, we will investigate how information exchange strategies such as clause sharing can be integrated more deeply into our procedures. Since parameters governing such information exchange are global (rather than restricted to an individual component solver), we will also investigate improved methods for handling global portfolio parameters. Finally, we will consider ways of reusing already trained portfolios for building new ones, for instance, if the instance set changes slightly or new solvers become available.

4 Algorithm Scheduling via Answer Set Programming

Boolean Constraint Technology has made tremendous progress over the last decade, leading to industrial-strength solvers. Although this advance in technology was mainly conducted in the area of Satisfiability Testing (SAT; (Biere et al., 2009)), it meanwhile also led to significant boosts in neighboring areas, like Answer Set Programming (ASP; (Baral, 2003)), Pseudo-Boolean Solving (Biere et al., 2009, Chapter 22), and even (multi-valued) Constraint Solving (Tamura et al., 2009). However, there is a prize to pay. Modern Boolean constraint solvers are rather sensitive to the way their search parameters are configured. Depending on the choice of the respective configuration, the solver's performance may vary by several orders of magnitude. Although this is a well-known issue, it was impressively illustrated once more during the 2011 SAT Competition, where 16 prizes were won by the portfolio-based solver *ppfolio* (Roussel, 2011). The idea underlying *ppfolio* is very simple: it independently runs several solvers in parallel. If only one processing unit is available, three solvers are started. By relying on the process scheduling mechanism of the operating system, each solver gets nearly the same time to solve a given instance. We refer to this as a uniform, unordered algorithm schedule[1]. If several processing units are available, one solver is started on each unit; however, multiple solvers may end up on the last unit.

Inspired by this simple, yet effective system, we devise a more elaborate, yet still simple approach that takes advantage of the modeling and solving capacities of ASP to automatically determine more refined, that is, non-uniform and ordered algorithm schedules from existing benchmarking data. The resulting encodings are easily customizable for different settings. For instance, our approach is directly extensible to the generation of parallel schedules for multi-processor machines. Also, the computation of optimum schedules can mostly be done in the blink of an eye, even when dealing with large runtime data sets stemming from many algorithms on hundreds to thousands of instances. Despite its simplicity, our approach matches the performance of much more sophisticated ones, such as *SATzilla* (Xu et al., 2008) and *3S* (Kadioglu et al., 2011). Unlike both, our approach does not rely on the availability of domain-specific features of the problem instance being solved, which makes it easily adaptable to other domains.

The remainder of this article is structured as follows. In Section 4.1, we formulate the problem of determining optimum schedules as a multi-criteria optimization problem. In doing so, our primary emphasis lies in producing robust schedules that aim at the fewest number of timeouts by non-uniformly attributing each algorithm (or algorithm configuration) a different time slice. Once such a robust schedule is found, we optimize its runtime by selecting the best algorithm

[1]We refer to algorithms here as a more general concept as solvers since schedules can be applied to arbitrary algorithms and not only to solvers. However, in the context of this chapter, algorithm can be synonymously understood as solvers.

	a_1	a_2	a_3	*oracle*
i_1	**1**	≥ 10	3	1
i_2	5	≥ 10	**2**	2
i_3	8	**1**	≥ 10	1
i_4	≥ 10	≥ 10	**2**	2
i_5	≥ 10	**6**	≥ 10	6
i_6	≥ 10	8	≥ 10	8
timeouts	3	3	3	0

Table 4.1: Table of algorithm runtimes on problem instances with $t_c = 10$; '≥ 10' indicates a timeout.

alignment. We then extend this approach to parallel settings in which multiple processing units are available. With these formalizations at hand, we proceed in two steps. First, we provide an ASP encoding for computing (parallel) timeout-minimal schedules (Section 4.2). Once such a schedule is identified, we use a second encoding to find a time-minimal alignment of its algorithms (Section 4.3). Both ASP encodings are also of interest from an ASP modelling perspective, because they reflect interesting features needed for dealing with large sets of (runtime) data. Finally, in Section 4.4, we provide an empirical evaluation of the resulting system *aspeed*, and we contrast it with related approaches (Section 4.5). In what follows, we presuppose a basic acquaintance with ASP (see (Gebser et al., 2012) for a comprehensive introduction).

4.1 Algorithm Scheduling

In the following, we formulate the optimization problem of computing an algorithm schedule. To this end, we introduce robust timeout-minimal schedules for single-threaded systems that are extended by an algorithm alignment mechanism to minimize the used runtime. Furthermore, in order to exploit the increasing prevalence of multi-core processors, we consider the problem of finding good parallel algorithm schedules.

4.1.1 Sequential Scheduling

Given a set I of problem instances and a set A of algorithms (for example, solvers with a fixed configuration), we use function $t : I \times A \mapsto \mathbb{R}^+$ to represent a table of algorithm runtimes on instances. Also, we use an integer t_c to represent a given cutoff time. For illustration, consider the runtime function in Table 4.1; it deals with 6 problem instances, i_1 to i_6, and 3 algorithms, a_1, a_2, and a_3.

Each algorithm can solve three out of six instances within the cutoff time, $t_c = 10$; timeouts are indicated by '≥ 10' in Table 4.1. The oracle, also known as virtual best solver (VBS), is obtained by assuming the best performance of each individual algorithm. As we see in the rightmost column, the oracle would be able to solve all instances in our example within the cutoff time; thus, if we knew beforehand which algorithm to choose for each instance, we could solve all of them. While we can hardly hope to practically realize an oracle on a single threaded system (at least in terms of CPU time), performance improvements can already be obtained by

successively running each algorithm for a limited period of time rather than running a single algorithm until the cutoff is reached. For instance, by uniformly distributing time over all three algorithms in our example, as done in *ppfolio*, we could solve 4 out of 6 instances, namely instance $i_1 \dots i_4$. Furthermore, the number of solved instances can be increased further by running a_1 for 1, a_2 for 6, and a_3 for 2 seconds, which allows us to solve 5 out of 6 instances, as indicated in bold in Table 4.1. In what follows, we show how such a optimized non-uniform schedule can be obtained beforehand from given runtime data.

Given I, A, t, and t_c as specified above, a *timeout-optimal algorithm schedule* can be expressed as a function $\sigma : A \to [0, t_c]$, satisfying the following condition:

$$\begin{aligned} \sigma \in \arg\max_{\sigma:A\to[0,t_c]} & \left|\{i \mid \exists a \in A : t(i,a) \leq \sigma(a)\}\right| \\ \text{such that} \quad & \textstyle\sum_{a\in A}\sigma(a) \leq t_c \end{aligned} \tag{4.1}$$

An optimal schedule σ consists of slices $\sigma(s)$ indicating the (possibly zero) time allotted to each algorithm $a \in A$. Such a schedule maximizes the number of solved instances, or conversely, minimizes the number of timeouts. An instance i is solved by σ if there is an algorithm $a \in A$ that has an equal or greater time slice $\sigma(a)$ than the time needed by the algorithm to solve the instance, viz. $t(i,a)$. As a side constraint, the sum of all time slices $\sigma(a)$ has to be equal or less than the cutoff time t_c.

The above example corresponds to the schedule $\sigma = \{a_1 \mapsto 1, a_2 \mapsto 6, a_3 \mapsto 2\}$; in fact, σ constitutes one of nine timeout-optimal algorithm schedules in our example. Note that the sum of all time slices is even smaller than the cutoff time. Hence, all schedules obtained by adding 1 to either of the three algorithms are also timeout-optimal. A timeout-optimal schedule consuming the entire allotted time is $\{a_1 \mapsto 0, a_2 \mapsto 8, a_3 \mapsto 2\}$.

In practice, however, the criterion in (4.1) turns out to be too coarse, that is, it often admits a diverse set of solutions among which we would like to make an educated choice. To this end, we make use of (simplified) L-norms as the basis for refining our choice of schedule. In our case, an L^n-norm on schedules is defined[2] as $\Sigma_{a\in A,\sigma(a)\neq 0}\sigma(a)^n$. Depending on the choice of n as well as whether we minimize or maximize the norm, we obtain different selection criteria. For instance, L^0-norms suggest using as few (or as many) algorithms as possible, and L^1-norms aim at minimizing (or maximizing) the sum of time slices. Minimizing the L^2-norm amounts to allotting each algorithm a similar time slice, while maximizing it prefers schedules with large runtimes for few algorithms. In more formal terms, for a given set A of algorithms, using an L^n-norm we would like to determine schedules satisfying the constraint

$$\sigma \in \arg\min_{\sigma:A\to[0,t_c]} \Sigma_{a\in A,\sigma(a)\neq 0}\sigma(a)^n, \tag{4.2}$$

or the analogous constraint for arg max (in case of maximization).

For instance, our example schedule $\sigma = \{a_1 \mapsto 1, a_2 \mapsto 6, a_3 \mapsto 2\}$ has the L^n-norms 3, 9, and 41 for $n = 0..2$. In contrast, we obtain norms 3, 9, and 27 for the (suboptimal) uniform schedule $\{a_1 \mapsto 3, a_2 \mapsto 3, a_3 \mapsto 3\}$ and 1, 9, and 81 for a singular schedule $\{a_3 \mapsto 9\}$, respectively. Although empirically, we found that schedules for various n as well as for minimization and maximization have useful properties, overall, we favor schedules with a minimal L^2-norm. First,

[2] The common L^n-norm is defined as $\sqrt[n]{\Sigma_{x\in X} x^n}$. We take the simpler definition in view of using it merely for optimization.

this choice leads to a significant reduction of candidate schedules and, second, it results in schedules with a maximally homogeneous distribution of time slices, similar to *ppfolio*. In fact, our example schedule has the smallest L^2-norm among all nine timeout-optimal algorithm schedules.

Once we have identified an optimal schedule w.r.t. criteria (4.1) and (4.2), it is interesting to determine which algorithm alignment yields the best performance as regards time. More formally, we define an *alignment* of a set A of algorithms as a bijective function $\pi : \{1,\ldots,|S|\} \to S$. Consider the above schedule $\sigma = \{a_1 \mapsto 1, a_2 \mapsto 6, a_3 \mapsto 2\}$. The alignment $\pi = \{1 \mapsto a_1, 2 \mapsto a_3, 3 \mapsto a_2\}$ induces the execution sequence (a_1, a_3, a_2) of σ. This sequence takes 29 seconds for all six benchmarks in Table 4.1; in detail, it takes $1, 1+2, 1+2+1, 1+2, 1+2+6, 1+2+7$ seconds for benchmark i_k for $k = 1..6$, whereby instance i_6 could not be solved. For instance, benchmark i_3 is successfully solved by the third algorithm in the alignment, viz. a_2. Hence the total time amounts to the time allotted by σ to a_1 and a_3, viz. $\sigma(a_1)$ and $\sigma(a_3)$, plus the effective time of a_2, viz. $t(i_3, a_2)$.

This can be formalized as follows. Given a schedule σ and an alignment π of a set A of algorithms, and an instance $i \in I$, we define the runtime τ of schedule σ aligned by π on i:

$$\tau_{\sigma,\pi}(i) = \begin{cases} \left(\sum_{j=1}^{\min(P_{\sigma,\pi})-1} \sigma(\pi(j))\right) + t(i, \pi(\min(P_{\sigma,\pi}))) & \text{if } P_{\sigma,\pi} \neq \emptyset, \\ t_c & \text{otherwise} \end{cases} \tag{4.3}$$

where $P_{\sigma,\pi} = \{l \in \{1,\ldots,|A|\} \mid t(i, \pi(l)) \leq \sigma(\pi(l))\}$ are the positions of algorithms solving instance i in a schedule σ aligned by π. If an instance i cannot be solved at all by a schedule, $\tau_{\sigma,\pi}(i)$ is set to the cutoff t_c. For our example schedule σ and its alignment π, we obtain for i_3: $\min P_{\sigma,\pi} = 3$ and $\tau_{\sigma,\pi}(i_3) = 1+2+1 = 4$.

For a schedule σ of algorithms in A, we then define the optimal alignment of schedule σ:

$$\pi \in \arg\min_{\pi:\{1,\ldots,|A|\}\to A} \sum_{i\in I} \tau_{\sigma,\pi}(i) \tag{4.4}$$

For our timeout-optimal schedule $\sigma = \{a_1 \mapsto 1, a_2 \mapsto 6, a_3 \mapsto 2\}$ w.r.t. criteria (4.1) and (4.2), we obtain two optimal execution alignments, namely (a_3, a_1, a_2) and (a_1, a_3, a_2), both of which result in a solving time of 29 seconds for the benchmarks of Table 4.1.

4.1.2 Parallel Scheduling

The increasing availability of multi-core processors makes it interesting to extend our approach for distributing schedule's algorithms over multiple processing units. For simplicity, we take a coarse approach in binding algorithms to units, thus precluding re-allocations during runtime.

To begin with, let us provide a formal specification of the extended problem. To this end, we augment our previous formalization with a set U of (processing) units and associate each unit with subsets of algorithms from A. More formally, we define a *distribution* of a set A of algorithms as the function $\eta : U \to 2^A$ such that $\bigcap_{u\in U} \eta(u) = \emptyset$. With it, we can determine timeout-optimal algorithm schedules for several cores simply by strengthening the condition in (4.1) to the effect that all algorithms associated with the same unit must respect the cutoff

time. This leads us to the following extension of (4.1):

$$\begin{aligned} &\sigma \in \arg\max_{\sigma:A\to[0,t_c]} |\{i \mid \exists a \in A : t(i,a) \leq \sigma(a)\}| \\ &\text{such that} \qquad \textstyle\sum_{a\in\eta(u)}\sigma(a) \leq t_c \;\text{ for each } u \in U \end{aligned} \tag{4.5}$$

For illustration, let us reconsider Table 4.1 along with schedule $\sigma = \{a_1 \mapsto 1, a_2 \mapsto 8, a_3 \mapsto 2\}$. Assume that we have two cores, 1 and 2, along with the distribution $\eta = \{1 \mapsto \{a_2\}, 2 \mapsto \{a_1, a_3\}\}$. This distributed schedule is an optimal solution to the optimization problem in (4.5) w.r.t. the benchmarks in Table 4.1 because it solves all benchmarks within a cutoff time of $t_c = 8$.

We keep the definitions of a schedule's L^n-norm as a global constraint. However, for determining our secondary criterion, enforcing time-optimal schedules, we relativize the auxiliary definitions in (4.3) to account for each unit separately. Given a schedule σ and a set U of processing units, we define for each unit $u \in U$ a *local alignment* of the algorithms in $\eta(u)$ as the bijective function $\pi_u : \{1, \ldots, |\eta(u)|\} \to \eta(u)$. Given this function and a problem instance $i \in I$, we extend the definitions in (4.3) as follows:

$$\tau_{\sigma,\pi_u}(i) = \begin{cases} \left(\sum_{j=1}^{\min(P_{\sigma,\pi})-1} \sigma(\pi_u(j))\right) + t(i, \pi_u(\min(P_{\sigma,\pi}))) & \text{if } P_{\sigma,\pi} \neq \emptyset, \\ t_c & \text{otherwise} \end{cases} \tag{4.6}$$

where $P_{\sigma,\pi} = \{l \in \{1, \ldots, |\eta(u)|\} \mid t(i, \pi_u(l)) \leq \sigma(\pi_u(l))\}$.

The collection $(\pi_u)_{u\in U}$ regroups all local alignments into a *global alignment*. For a schedule σ of algorithms in A and a set U of (processing) units, we then define an optimal global alignment:

$$(\pi_u)_{u\in U} \in \arg\min_{(\pi_u:\{1,\ldots,|\eta(u)|\}\to\eta(u))_{u\in U}} \textstyle\sum_{i\in I} \min_{u\in U} \tau_{\sigma,\pi_u}(i) \tag{4.7}$$

For illustration, reconsider the above schedule $\sigma = \{a_1 \mapsto 1, a_2 \mapsto 8, a_3 \mapsto 2\}$ and distribution $\eta = \{1 \mapsto \{a_2\}, 2 \mapsto \{a_1, a_3\}\}$, and suppose we chose the local alignments $\pi_1 = \{1 \mapsto a_2\}$ and $\pi_2 = \{1 \mapsto a_1, 2 \mapsto a_3\}$. This global alignment solves all six benchmark instances of Table 4.1 in 22 seconds wallclock time. In more detail, it takes $1_2, 1+2_2, 1_1, 1+2_2, 6_1, 8_1$ seconds for instance i_k for $k = 1..6$, where the solving unit is indicated by the subscript.

Note that the definitions in (4.5), (4.6), and (4.7) correspond to their sequential counterparts in (4.1), (4.3), and (4.4) whenever we are faced with a single processing unit.

4.2 Solving Timeout-Optimal Scheduling with ASP

To begin with, we detail the basic encoding for identifying robust (parallel) schedules. In view of the remark at the end of the last section, however, we directly provide an encoding for parallel scheduling, which collapses to one for sequential scheduling whenever a single processing unit is used.

Following good practice in ASP, a problem instance is expressed as a set of facts. That is, Function $t : I \times A \mapsto \mathbb{R}$ is represented as facts of form `time(i,a,t)`, where `i` $\in I$, `a` $\in A$, and `t` is the runtime `t(i,a)`, converted to a natural number with limited precision. The cutoff is expressed via Predicate `cutoff/1`, and the number of available processing units is captured via Predicate `units/1`, here instantiated to 2 units. Given this, we can represent the contents of

Table 4.1 as shown in Listing 4.1 below.

```
cutoff(10).
units(2).

time(i1, a1,  1).  time(i1, a2, 11).  time(i1, a3,  3).
time(i2, a1,  5).  time(i2, a2, 11).  time(i2, a3,  2).
time(i3, a1,  8).  time(i3, a2,  1).  time(i3, a3, 11).
time(i4, a1, 11).  time(i4, a2, 11).  time(i4, a3,  2).
time(i5, a1, 11).  time(i5, a2,  6).  time(i5, a3, 11).
time(i6, a1, 11).  time(i6, a2,  8).  time(i6, a3, 11).
```

Listing 4.1: Facts

The encoding in Listing 4.3 along with all following ones are given in the input language of *gringo* (Gebser, Kaminski, Kaufmann, Ostrowski, Schaub, & Thiele,). The first three lines of Listing 4.3 provide auxiliary data. The set A of algorithms is given by Predicate `algorithm/1`. Similarly, the runtimes for each algorithm are expressed by `time/2` and each processing unit by `unit/1`. In addition, the ordering of instances by time per algorithm is precomputed; it is expressed via `order/3`, as shown in Figure 4.2.

```
order(I,J,A) :-
 time(I,A,T), time(J,A,V), (T,I) < (V,J),
 not time(K,A,U) : time(K,A,U) : (T,I) < (U,K) : (U,K) < (V,J).
```

Listing 4.2: `I` is solved immediatly before `J` by algorithm `A`

The above results in facts `order(I,J,A)` capturing that instance `J` follows instance `I` by sorting the instances according to their runtimes. Although this information could be computed via ASP (as shown above), we make use of external means for sorting (the above rule needs cubic time for instantiation, which is infeasible for a few thousand instances). Instead, we use *gringo*'s embedded scripting language *lua* for sorting.

The idea of Listing 4.3 is now to guess for each algorithm a time slice and a processing unit (in Line 5). With the resulting schedule, all solvable instances can be identified (in Line $10-12$), and finally, all schedules solving a maximal number of instances are selected (in Line 14).

In more detail, a schedule is represented by atoms `slice(U,A,T)` allotting a time slice `T` to algorithm `A` on unit `U`. In Line 5, at most one time slice is chosen for each algorithm, subject to the condition that it does not exceed the cutoff time. At the same time, a processing unit is uniquely assigned to the selected algorithm. The integrity constraint in Line 6 ensures that the sum over all selected time slices on each processing unit is not greater than the cutoff time. This implements the side condition in (4.5), and it reduces to the one in (4.1) whenever a single unit is considered. The next line projects out the processing unit because it is irrelevant when determining solved instances (in Line 8). In Lines 10 to 12, all instances solved by the selected time slices are gathered via predicate `solved/1`. Considering that we collect in Line 8 all time slices among actual runtimes, each time slice allows for solving at least one instance. This property is used in Line 10 to identify the instance `I` solvable by algorithm `A`; using it, along with the sorting of instances by algorithm performance in `order/3`, we collect in Line 11 all

```
1  algorithm(A)  :- time(_,A,_).
2  time(A,T)  :- time(_,A,T).
3  unit(1..N) :- units(N).
4
5  {slice(U,A,T): time(A,T): T <= K: unit(U)} 1 :- algorithm(A), cutoff(K).
6   :- not [ slice(U,A,T) = T ] K, cutoff(K), unit(U).
7
8  slice(A,T) :- slice(_,A,T).
9
10 solved(I,A) :- slice(A,T), time(I,A,T).
11 solved(I,A) :- solved(J,A), order(I,J,A).
12 solved(I)   :- solved(I,_).
13
14 #maximize { solved(I) @ 2 }.
15 #minimize [ slice(A,T) = T*T @ 1 ].
```

Listing 4.3: ASP encoding for Timeout-Minimal (Parallel) Scheduling

instances that can be solved even faster than the instance in Line 10. Note that at first sight it might be tempting to encode Lines 10 – –12 differently:

```
solved(I) :- slice(A,T), time(I,A,TS), T <= TS.
```

The problem with the above rule is that it has a quadratic number of instantiations in the number of benchmark instances in the worst case. In contrast, our ordering-based encoding is linear, because only successive instances are considered. Finally, the number of solved instances is maximized in Line 14, using the conditions from (4.5) (or (4.1), respectively). This primary objective is assigned a higher priority than the L^2-norm from (4.2) (priority 2 *vs* 1).

4.3 Solving (Timeout and) Time-Minimal Parallel Scheduling with ASP

In the previous section, we have explained how to determine a timeout-minimal (parallel) schedule. Here, we present an encoding that takes such a schedule and calculates an algorithm alignment per processing unit while minimizing the overall runtime according to Criterion (4.7). This two-phase approach is motivated by the fact that an optimal alignment must be determined among all permutations of a schedule. While a one-shot approach had to account for all permutations of all potential timeout-minimal schedules, our two-phase approach reduces the second phase to searching among all permutations of a single timeout-minimal schedule.

We begin by extending the ASP formulation from the last section (in terms of `cutoff/1`, `units/1`, and `time/3`) by facts over `slice/3` providing the time slices of a timeout-minimal schedule (per algorithm and processing unit). In the case of our example from Section 4.1.2, we extend the facts of Listing 4.1 with the following obtained timeout-minimal schedule to create the problem instance:

```
slice(1,a2,8). slice(2,a1,1). slice(2,a3,2).
```

Listing 4.4: Schedule Facts

The idea of the encoding in Listing 4.5 is to guess a permutation of algorithms and then to use ASP's optimization capacities for calculating a time-minimal alignment. The challenging part is to keep the encoding compact. That is, we have to keep the size of the instantiation of the encoding small, because otherwise, we cannot hope to effectively deal with rather common situations involving thousands of benchmark instances. To this end, we make use of `#sum` aggregates with negative weights (Line 23) to find the fastest processing unit without representing any sum of times explicitly.

```
1  algorithm(U,A)  :- slice(U,A,_).
2  instance(I)     :- time(I,_,_).
3  unit(1..N)      :- units(N).
4  algorithms(U,N)    :- unit(U), N := {algorithm(U,_)}.
5  solved(U,A,I)   :- time(I,A,T), slice(U,A,TS), T <= TS.
6  solved(U,I)     :- solved(U,_,I).
7  capped(U,I,A,T) :- time(I,A,T), solved(U,A,I).
8  capped(U,I,A,T) :- slice(U,A,T), solved(U,I), not solved(U,A,I).
9  capped(U,I,d,K) :- unit(U), cutoff(K), instance(I), not solved(U,I).
10 capped(I,A,T)   :- capped(_,I,A,T).
11
12 1 { order(U,A,X) : algorithm(U,A) } 1 :- algorithms(U,N), X = 1..N.
13 1 { order(U,A,X) : algorithms(U,N) : X = 1..N } 1 :- algorithm(U,A).
14
15 solvedAt(U,I,X+1) :- solved(U,A,I), order(U,A,X).
16 solvedAt(U,I,X+1) :- solvedAt(U,I,X), algorithms(U,N), X <= N.
17
18 mark(U,I,d,K) :- capped(U,I,d,K).
19 mark(U,I,A,T) :- capped(U,I,A,T), order(U,A,X), not solvedAt(U,I,X).
20 min(1,I,A,T)  :- mark(1,I,A,T).
21
22 less(U,I) :- unit(U), unit(U+1), instance(I),
23   [min(U,I,A1,T1): capped(I,A1,T1) = T1, mark(U+1,I,A2,T2) = -T2] 0.
24
25 min(U+1,I,A,T) :- min(U,I,A,T), less(U,I).
26 min(U,I,A,T)   :- mark(U,I,A,T), not less(U-1,I).
27
28 #minimize [min(U,_,_,T): not unit(U+1) = T].
```

Listing 4.5: ASP encoding for Time-Minimal (Parallel) Scheduling

The block in Line 1 to 10 gathers static knowledge about the problem instance, that is, algorithms per processing unit (`algorithm/2`), instances appearing in the problem description (`instance/1`), available processing units (`unit/1`), number of algorithms per unit (`algorithms/2`), instances solved by an algorithm within its allotted slice (`solved/3`), and instances that could be solved on a unit given the schedule (`solved/2`). Note that, in contrast to the previous encoding (Listing 4.3), the solved instances (`solved/3`) can be efficiently expressed as done in Line 5 of Listing 4.5, because `slice/3` are facts here. In view of Equation (4.6), we precompute the times that contribute to the values of τ_{σ,π_u} and capture them in `capped/4` (and `capped/3`). A fact `capped(U,I,S,T)` assigns to instance `I` run by algorithm `A` on unit `U` a time `T`. In Line 7, we assign the time needed to solve the instance if it is within the algorithm's time slice. In Line 8, we assign the algorithm's time slice if the instance could not be solved, but at least one other

algorithm could solve it on processing unit `U`. In Line 9, we assign the entire cutoff to dummy algorithm `d` (we assume that there is no other algorithm called `d`) if the instance could not be solved on the processing unit at all; this is to implement the else case in (4.6) and (4.3).

The actual encoding starts in Line 12 and 13 by guessing a permutation of algorithms. Here, the two head aggregates ensure that for every algorithm (per unit) there is exactly one position in the alignment and vice versa. In Line 15 and 16, we mark indexes (per unit) as solved if the algorithm with the preceding index could solve the instance or if the previous index was marked as solved. Note that this is a similar "chain construction" used in the previous section in order to avoid a combinatorial blow-up.

In the block from Line 18 to 26, we determine the time for the fastest processing unit depending on the guessed permutation. The rules in Line 18 and 19 mark the times that have to be added up on each processing unit; the sums of these times correspond to $\tau_{\sigma,\pi_u}(i)$ in Equation (4.6) and (4.3). Next, we determine the smallest sum of times by iteratively determining the minimum. An atom `min(U,I,A,T)` marks the times of the fastest unit in the range from unit `1` to `U` to solve an instance (or the cutoff via dummy algorithm `d`, if the schedule does not solve the instance for the unit). To this end, we initialize `min/4` with the times for the first unit in Line 20. Then, we add a rule in Line 22 and 23 that, given minimal times for units in the range of `1` to `U` and times for unit `U+1`, determines the faster one. The current minimum contributes positive times to the sum, while unit `U+1` contributes negative times. Hence, if the sum is negative or zero, the sum of times captured in `min/4` is smaller than or equal to the sum of times of unit `U+1`, and therefore, the unit thus slower than some preceding unit, which makes the aggregate true and derives the corresponding atom over `less/2`. Depending on `less/2`, we propagate the smaller sum, which is either contributed by unit `U+1` (Line 25) or the preceding units (Line 26). Finally, in Line 28, the times of the fastest processing unit are minimized in the optimization statement, which implements Equation (4.7) and (4.4).

4.4 Empirical Performance Analysis

After describing the theoretical foundations and ASP encodings underlying our approach, we now present the results from an empirical evaluation on representative ASP, CSP, MaxSAT, SAT and QBF benchmarks. The python implementation of our approach, dubbed *aspeed*, uses the state-of-the-art ASP systems (Calimeri, Ianni, Ricca, Alviano, Bria, Catalano, Cozza, Faber, Febbraro, Leone, Manna, Martello, Panetta, Perri, Reale, Santoro, Sirianni, Terracina, & Veltri, 2011b) of the potassco group (Gebser et al., 2011), namely the grounder *gringo* (3.0.4) and the ASP solver *clasp* (2.0.5). The sets of runtime data used in this work are freely available online. [3]

4.4.1 Experimental Setup

Our experiments are based on a set of runtime data obtained by running several algorithms (or algorithm configurations) on a set of benchmark instances (similar to Table 4.1). To provide a thorough empirical evaluation of our approach, we selected eight large data sets of runtimes for five prominent and widely studied problems, ASP, CSP, MaxSAT, SAT and QBF; these are summarized in Table 4.2. The sets *Random*, *Crafted* and *Application* contain the authentic

[3] `http://www.cs.uni-potsdam.de/aspeed`

	Random	*Crafted*	*Application*	*ASP-Set*
Cutoff (sec.)	5000	5000	5000	900
#Instances	600	300	300	2589
#Algorithms	9	15	18	25
Source	(1)	(1)	(1)	(2)

	3S-Set	*CSP-Set*	*QBF-Set*	*MaxSAT-Set*
Cutoff (sec.)	5000	5000	3600	1800
#Instances	5467	2024	1368	337
#Algorithms	37	2	5	11
Source	(3)	(4)	(5)	(6)

Table 4.2: Runtime data sets used in our experiments from the 2011 SAT Competition (1), the ASP benchmark repository *asparagus* (2), Kadioglu et al. 2011 (3), Gent et al. 2010 (4), Pulina and Tacchella 2009 (5) and Malitsky et al. 2013 (6).

runtimes taken from the 2011 SAT Competition[4] with a cutoff of 5000 seconds. We selected all non-portfolio, non-parallel solvers from the main phase of the competition, in order to provide a fair comparison with the portfolio-based SAT Solver *SATzilla* (Xu et al., 2008), which has been evaluated based on the same data (Xu et al., 2012a).

Also, we evaluated our approach on an ASP instance set (*ASP-Set*) based on different configurations of the highly parametric ASP solver *clasp* (Gebser, Kaufmann, & Schaub, 2012b), which is known to show excellent performance on a wide range of ASP instances. We used the complementary configuration portfolio of *claspfolio* (1.0.1) (Gebser et al., 2011) designed by the main developer of *clasp*, B. Kaufmann, and measured the runtime of *clasp* (2.1.0). Because the instance sets from recent ASP competitions are very unbalanced (Hoos et al., 2013) (most of them are either too easy or too hard for *clasp*), we select instances from the ASP benchmark repository *Asparagus*,[5] including the 2007 (SLparse track), 2009 and 2011 ASP Competitions. *gringo* was not able to ground some instance from the 2011 ASP Competition within 600 CPU seconds and 2 GB RAM, and thus those instances were excluded. Our *ASP-Set* is comprised of the 2589 remaining instances.

The runtime measurements for our *ASP-Set* were performed on a compute cluster with 28 nodes, each equipped with two Intel Xeon E5520 2.26GHz quad-core CPUs and 48 GB RAM, running Scientific Linux (2.6.18-308.4.1.el5). Since all *clasp* configurations used in our experiments are deterministic, their runtimes on all instances were measured only once.

Furthermore, we evaluated our approach on sets already used in the literature. The set of runtime data provided by Kadioglu et al. was part of the submission of their solver *3S* (Kadioglu et al., 2011) to the 2011 SAT Competition. We selected this set, which we refer to as *3S-Set*, because it includes runtimes of many recent SAT solvers on prominent SAT benchmark instances. The *CSP-Set* was used by Gent, Jefferson, Kotthoff, Miguel, Moore, Nightingale, and Petrie (2010), the *QBF-Set* by Pulina and Tacchella (2009), and *MaxSAT-Set* by Malitsky, Mehta, and O'Sullivan (2013), respectively.

[4] `http://www.cril.univ-artois.fr/SAT11`

[5] `http://asparagus.cs.uni-potsdam.de`

The performance of *aspeed* was determined from the schedules computed for Encodings 4.3 and 4.5 with a minimization of the L^2-norm as second optimization criterion. Although we empirically observed no clear performance gain from the latter, we favour a schedule with a minimal L^2-norm: First, it leads to a significant reduction of candidate schedules and second, it results in schedules with a more uniform distribution of time slices, (resembling those used in *ppfolio*). All runtimes for the schedule computation were measured in CPU time rounded up to the next integer value, and runtime not allocated in the computed schedule was uniformly distributed among all algorithms in the schedule.

Using the previously described data sets, we compared *aspeed* against

- *single best*: the best algorithm in the respective portfolio,
- *uniform*: a uniform distribution of the time slices over all algorithms in the portfolio,
- *ppfolio-like*: an approach inspired by *ppfolio*, where the best three complementary algorithms are selected with an uniform distribution of time slices in the sequential case,
- *SATzilla* (Xu et al., 2012a) and *claspfolio* (Gebser et al., 2011), prominent examples of model-based algorithm selection solvers for SAT and ASP, respectively,
- as well as against the *oracle* performance (also called virtual best solver)[6].

The performance of *SATzilla* for *Random*, *Crafted* and *Application* was extracted from results reported in the literature (Xu et al., 2012a), which were obtained using 10-fold cross validation. In the same way, *claspfolio* was trained and cross-validated on the *ASP-Set*. In the following, the *selection* approach represents *SATzilla* for the three SAT competition sets and *claspfolio* for the *ASP-Set*.

Unfortunately, *aspeed* could not be directly compared against *3S*, because the tool used by *3S* to compute the underlying model is not freely available and hence, we were unable to train *3S* on new data sets. To perform a fair comparison between *aspeed* and *3S*, we compare both systems in an additional experiment in the last part of this section.

4.4.2 Schedule Computation

Table 4.3 shows the time spent on the computation and the proof of the optimality of timeout-minimal schedules and time-minimal alignments on the previously described benchmark sets for sequential schedules (first two rows) and parallel schedules for eight cores (next two rows). For the *Random*, *Crafted* and *CSP-Set* benchmark sets, the computation of the sequential and parallel schedule always took less than one CPU second. Some more time was spent for the *Application*, *QBF-Set* and *MaxSAT-Set* benchmark set but it is still feasible to find an optimal schedule. We observe that the computation of parallel time slices is faster than the computation of sequential schedules, except for the very simple *CSP-Set*. Given the additional processing units, the algorithms can be scheduled more freely, resulting in a less constrained problem that is easier to solve. Furthermore, calculating a time-minimal alignment is easier in the parallel setting. In our experiments, we obtained fewer selected algorithms on the individual cores than

[6]The performance of the *oracle* is the minimal runtime of each instance given a portfolio of algorithms and corresponds to a portfolio-based solver with a perfect selection of the best algorithm for a given instance.

#cores	Opt. Step	*Random*	*Crafted*	*Application*	*ASP-Set*
1	Schedule (sec)	0.54	0.45	119.2	$> 1d$
1	Alignment (sec)	0.04	0.23	0.07	0.50
8	Schedule (sec)	0.28	0.05	61.65	$> 1d$
8	Alignment (sec)	0.02	0.006	0.07	0.50
1	Combined (sec)	$> 1d$	47175	$> 1d$	*MEMOUT*
		3S-Set	*CSP-Set*	*QBF-Set*	*MaxSAT-Set*
1	Schedule (sec)	$> 1d$	0.10	14.98	1.64
1	Alignment (sec)	$> 1d$	0.04	0.75	0.02
8	Schedule (sec)	$> 1d$	0.20	0.21	0.30
8	Alignment (sec)	$> 1d$	0.12	0.27	0.02
1	Combined (sec)	*MEMOUT*	0.89	32.09	$> 1d$

Table 4.3: Runtimes of *clasp* in CPU seconds to calculate an optimal schedule for one and eight cores.

in the sequential case. This leads to smaller permutations of algorithms and, in turn, reduces the total runtime. For the *ASP-Set*, we could not establish the optimal schedule even after one CPU day and for the *3S-Set*, the calculation of the optimal schedule and optimal alignment was also impossible. However *aspeed* was nevertheless able to find schedules and aligments, and hence, was able to minimize the number of timeouts and runtime. Finally, it is also possible that *aspeed* found an optimal schedule but was unable to prove its optimality. Therefore, we limited the maximal runtime of *clasp* for these sets to 1200 CPU seconds in all further experiments, and used the resulting sub-optimal schedules and alignments obtained for this time.[7]

We also ran experiments on an encoding that optimizes the schedule and alignment simultaneously; this approach accounts for all permutations of all potential timeout-minimal schedules. The results are presented in the row labelled 'Combined' in Table 4.3. The combination increases the solving time drastically. Within one CPU day, *clasp* was able to find an optimal solution and proved optimality only for *Crafted*, *CSP-Set* and *QBF-Set*. In all other cases, we aborted *clasp* after one CPU day and then used the best schedules found so far. Nevertheless, we could find better alignments than in our two step approach (between 0.6% and 9.8% improvement), at the cost of substantially higher computation time and memory. Because this encoding has a very large instantiation, viz., more than 12 GB memory consumption, we were unable to run *aspeed* using it on the *3S-Set* and *ASP-Set*.

4.4.3 Evaluation of Timeout-Minimal Schedules

Having established that optimal schedules can be computed within a reasonable time in most cases, we evaluated the sequential timeout-minimal schedule of *aspeed* corresponding to the first step of our optimization process (cf. Equation (4.1)). The number of timeouts for a fixed time budget assesses the robustness of an algorithm and is in many applications and competitions the primary evaluation criterion.

[7]Note that in our experiments, the performance of *unclasp* (Andres, Kaufmann, Matheis, & Schaub, 2012), which optimizes based on unsatisfiable cores, did not exceed the perfomance of *clasp* in computing algorithm schedules.

	Random	*Crafted*	*Application*	*ASP-Set*
single best	254/600	155/300	85/300	446/2589
uniform	155/600	123/300	116/300	536/2589
ppfolio-like	127/600	126/300	88/300	308/2589
selection	**115**/600	101/300	**74**/300	296/2589
aspeed	131/600	**98**/300	83/300	**290**/2589
oracle	108/600	77/300	45/300	156/2432
	3S-Set	*CSP-Set*	*QBF-Set*	*MaxSAT-Set*
single best	1881/5467	288/2024	579/1368	99/337
uniform	1001/5467	283/2024	357/1368	21/337
ppfolio-like	796/5467	283/2024	357/1368	10/337
aspeed	**603**/5467	**275**/2024	**344**/1368	**7**/337
oracle	0/5467	253/2024	314/1368	0/337

Table 4.4: Comparison of different approaches w.r.t. #timeouts / #instances. The performance of the best performing system is in boldface.

To obtain an unbiased evaluation of performance, we used 10-fold cross validation, a standard technique from machine learning: First, the runtime data for a given instance set are randomly divided into 10 equal parts. Then, in each of the ten iterations, 9/10th of the data is used as a training set for the computation of the schedule and the remaining 1/10th serves as a test set to evaluate the performance of the algorithm schedule at hand; the results shown are obtained by summing over the folds. We compared the schedules computed by *aspeed* against the performance obtained from the *single best*, *uniform*, *ppfolio-like*, *selection* (*SATzilla* and *claspfolio*; if possible) approaches and the (theoretical) *oracle*. The latter provides a bound on the best performance obtainable from any portfolio-based solver.

Table 4.4 shows the fraction of instances in each set on which timeouts occurred (smaller numbers indicate better performance). In all cases, *aspeed* showed better performance than the *single best* algorithm. For example, *aspeed* reduced the number of timeouts from 1881 to 603 instances (less 23% of unsolved instances) on the *3S-Set*, despite the fact that *aspeed* was unable to find the optimal schedule within the given 1200 CPU seconds on this set. Also, *aspeed* performed better than the *uniform* approach. The comparison with *ppfolio-like* and *selection* (*SATzilla* and *claspfolio*) revealed that *aspeed* performed better than *ppfolio-like* in seven out of eight scenarios we considered, and better than *SATzilla* and *claspfolio* in two out of four scenarios. We expected that *aspeed* would solve fewer instances than the *selection* approach in all four scenarios, because *aspeed*, unlike *SATzilla* and *claspfolio*, does not use any instance features or prediction of algorithm performance. It is somewhat surprising that *SATzilla* and *claspfolio* do not always benefit from their more sophisticated approaches, and further investigation into why this happens would be an interesting direction for future work.

4.4.4 Evaluation of Time-Minimal Alignment

After choosing the time slices for each algorithm, it is necessary to compute an appropriate algorithm alignment in order to obtain the best runtimes for our schedules. As before, we used

10-fold cross validation to assess this stage of *aspeed*. To the best of our knowledge, there is no system with a computation of alignments to compare against. Hence, we use a random alignment as a baseline for evaluating our approach. Thereby, the expected performance of a random alignment is the average runtime of all possible alignments. Since the number of all permutations for *ASP-Set* and *3S-Set* is too large ($\gg$ 1 000 000 000), we approximate the performance of a random alignment by 10 000 sampled alignments.

Table 4.5 shows the ratio of the expected performance of a random alignment and alignments computed by *aspeed*. Note that this ratio can be smaller than one, because the alignments are calculated on a training set and evaluated on a disjoint test set.

Also, we contrast the optimal alignment with two easily computable heuristic alignments to avoid the search for an optimal alignment. The alignment heuristic *heu-Opt* sorts algorithms beginning with the algorithm with the minimal number of timeouts (most robust algorithm), while *heu-Min* begins with the algorithm with the smallest time slice.

	Random	*Crafted*	*Application*	*ASP-Set**
aspeed	**1.16**	**1.15**	**1.03**	**1.13**
heu-Opt	1.02	0.84	1.00	1.05
heu-Min	1.15	1.14	1.00	1.12
	*3S-Set**	*CSP-Set*	*QBF-Set*	*MaxSAT-Set*
aspeed	**1.21**	**1.12**	**1.27**	**2.13**
heu-Opt	0.96	0.90	1.14	0.89
heu-Min	1.20	1.11	1.14	1.63

Table 4.5: Ratios of the expected performance of a random alignment and alignments computed by *aspeed*, *heu-Opt* and *heu-Min*; *heu-Opt* sorts the algorithms beginning with the algorithm with the minimal number of timeouts; *heu-Min* begins with the algorithm with the smallest time slice. The expected performance of a random alignment was approximated by 10.000 samples for all sets marked with *.

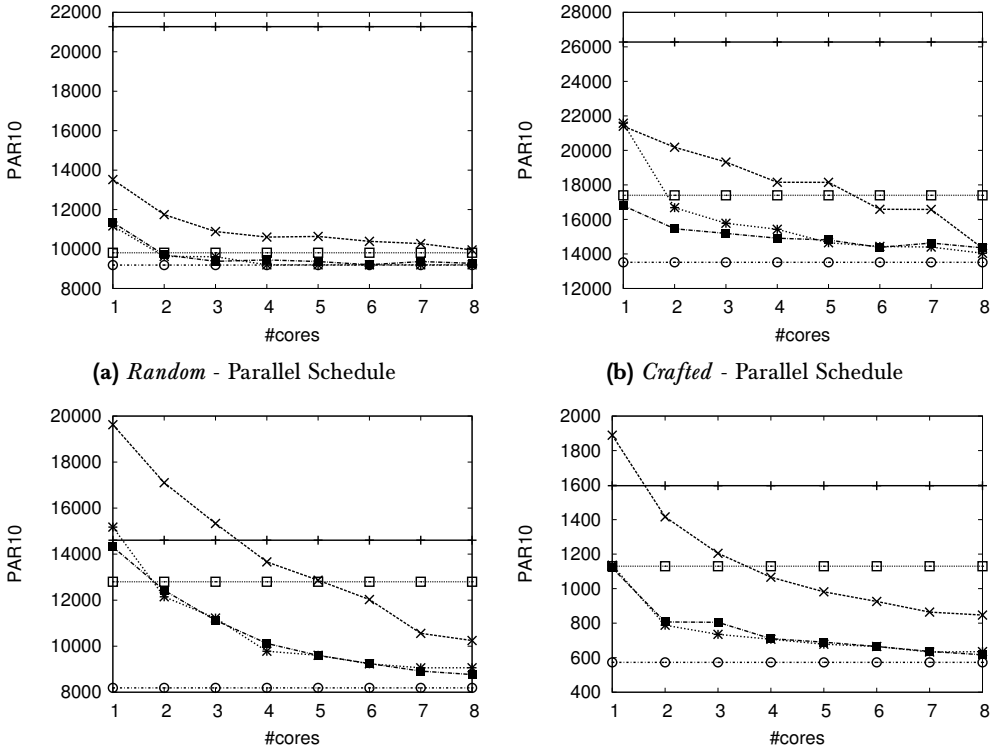
As expected, the best performance is obtained by using optimal alignments within *aspeed* (Table 4.5); it led, for example, to an increase in performance by a factor of 2.13 on *MaxSAT-Set*. In all cases, the performance of *heu-Min* was strictly better than (or equal to) that of *heu-Opt*. Therefore, using *heu-Min* seems desirable whenever the computation of an optimal alignment is infeasible.

The actual runtimes of *aspeed* and the other approaches are quite similar to the results on the number of timeouts (Table 4.4) (data not shown). The penalized runtimes (PAR10) are presented in Figure 4.1 (a),(b) and (c) at $\#cores = 1$.

4.4.5 Parallel Schedules

As we have seen in Section 4.2, our approach is easily extendable to parallel schedules. We evaluated such schedules on *Random*, *Crafted*, *Application* and *ASP-Set*. The results of this experiment are presented in Figure 4.1. These evaluations were performed using 10-fold cross validation and measuring wall-clock time.

In each graph, the number of cores is shown on the x-axis and the PAR10 (penalized average



(a) *Random* - Parallel Schedule

(b) *Crafted* - Parallel Schedule

(c) *Application* - Parallel Schedule

(d) *ASP-Set* - Parallel Schedule

Figure 4.1: Parallel Schedules *single best* (+), *uniform* (×), *ppfolio-like* approach (∗), *aspeed* (■), *selection* (□), *oracle* (◯).

runtime)[8] on the y-axis; we used PAR10, a commonly used metric from the literature, to capture average runtime as well as timeouts. (The sequential performance of *aspeed* can be read off the values obtained for one core.) Since the *single best* algorithm (+) and *selection* (□, *SATzilla* resp. *claspfolio*) cannot be run in parallel, their performance is constant. Furthermore, the *ppfolio-like* approach (∗) is limited to run at most three component algorithm on the first core with uniform time slices and one component algorithms on each other core. This more constrained schedule is also computed with the ASP encodings presented in Section 4.2 by adding three more constraints.

As stated previously, the sequential version of *aspeed* (■) performed worse than *SATzilla* (□) in *Random* and *Application*. However, *aspeed* turned out to perform at least as well as *SATzilla* when using two or more cores, in terms of PAR10 scores as well in terms of average runtime (data not shown). For example, *aspeed-4P* – that is parallel *aspeed* using four cores – achieved a speedup of 1.20 over the sequential *aspeed* on *Random* (20 fewer timeouts), 1.10 on *Crafted* (9 fewer timeouts), 1.44 on *Application* (26 fewer timeouts) and 1.57 on *ASP-Set* (111 fewer timeouts); furthermore, *aspeed-4P* solved 4, 13, 17, 117 instances more on these sets than (sequential) *SATzilla* and *claspfolio*, respectively. Considering the high performance

[8]PAR10 penalizes each timeout with 10 times the given cutoff time (Hutter et al., 2009).

of *SATzilla* (Xu et al., 2012a) and *claspfolio* (Gebser et al., 2011), this represents a substantial performance improvement.

	3S-Set		*CSP-Set*		*QBF-Set*		*MaxSAT-Set*	
	#TO	PAR10	#TO	PAR10	#TO	PAR10	#TO	PAR10
uniform-SP	1001	9847	283	7077	357	10176	21	1470
ppfolio-like-SP	796	7662	283	7077	357	9657	10	731
aspeed-SP	**603**	**6001**	**275**	**6902**	**344**	**9272**	**7**	**516**
uniform-4P	583	5720	**253**	**6344**	**316**	8408	4	511
ppfolio-like-4P	428	4095	**253**	**6344**	**316**	8404	4	353
aspeed-4P	**204**	**2137**	**253**	**6344**	**316**	**8403**	**3**	**332**
oracle	0	198	253	6344	314	8337	0	39

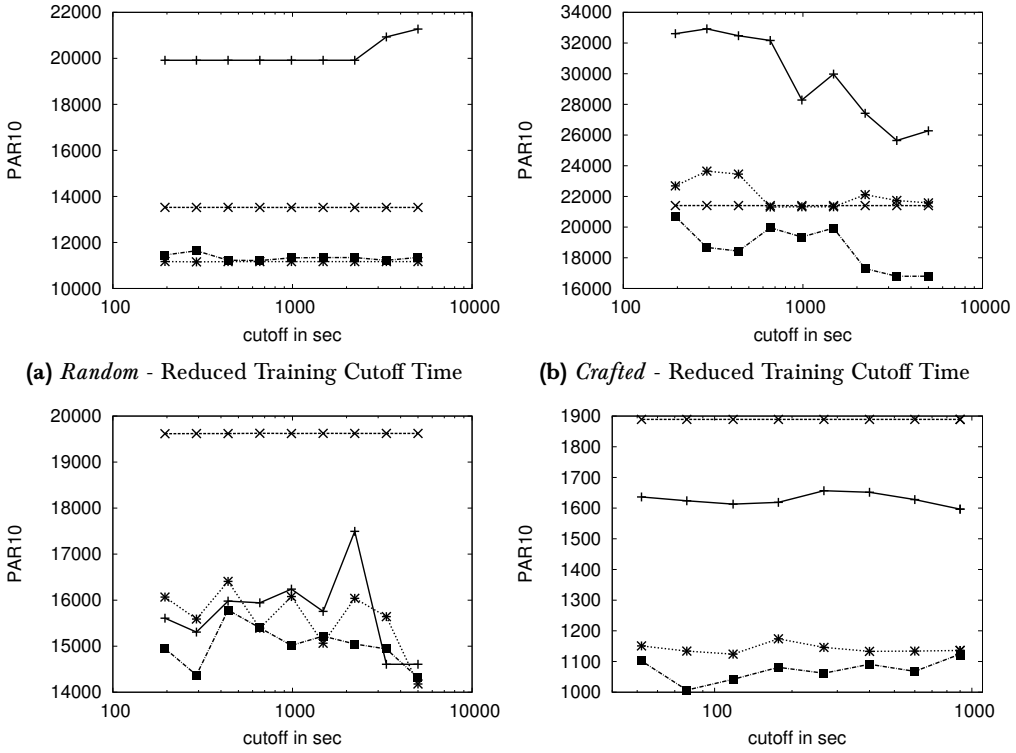
Table 4.6: Comparison of sequential and parallel schedules with 4 cores w.r.t. the number of timeouts and PAR10 score.

Table 4.6 presents the performance of parallel *aspeed* with four cores (*aspeed-4P*), the parallel *uniform* and parallel *ppfolio-like* schedule, respectively, on *3S-Set*, *CSP-Set*, *QBF-Set* and *MaxSAT-Set*. We decided to use only four cores because (i) *CSP-Set* and *QBF-Set* have two resp. five algorithms, and therefore it is trivial to perform as well as the *oracle* with 4 or more cores, and (ii) we saw in Figure 4.1 that the curves flatten beginning with four cores, which is an effect of the complementarity of the algorithms in the portfolio. The performance of *aspeed-SP*, that is, sequential *aspeed*, is already nearly as good as the *oracle* on *MaxSAT-Set* and *aspeed-4P* was only able to improve the performance slightly. However, *aspeed-4P* was able to decrease the number of timeouts from 603 to 204 on the *3S-Set*.

4.4.6 Generalization Ability of *aspeed*

The schedule computation of *aspeed* uses runtime data measurements, which require extensive computational resources. Therefore, we investigated the possibility to decrease the cutoff time on the training data to reduce the overall computational burden of training. The schedules thus obtained were evaluated on test data with an unreduced cutoff time. We note that only instances are considered for the computation of schedules that are solved by at least one algorithm in the portfolio. Therefore, using this approach with a lower training cutoff time, the computation of a schedule is based on easier and fewer instances than those in the test set used to ultimately evaluate it. Figures 4.2 show the results of evaluating the resulting schedules in the same way as in the experiments for parallel schedules with 10-fold cross validation but using only one processing unit. The cutoff time on the training set (shown on a logarithmic x-axis) was reduced according to a 2/3-geometric sequence, from the maximal cutoff time of 5000 down to 195 CPU seconds for *Random*, *Crafted* and *Application* and 900 down to 52 CPU seconds for the *ASP-Set*. A flat line corresponds to the expected optimal case that the performance of a schedule does not suffer from a reduced cutoff time; the *uniform* approach (×) does not rely on training data and therefore has such a constant performance curve.

Surprisingly, the reduced cutoff time had nearly no effect on the performance of *aspeed* (■) on *Random* (Figure 4.2a) and *ASP-Set* (Figure 4.2d). On the other hand, the selection of the *single*



(a) *Random* - Reduced Training Cutoff Time

(b) *Crafted* - Reduced Training Cutoff Time

(c) *Application* - Reduced Training Cutoff Time

(d) *ASP-Set* - Reduced Training Cutoff Time

Figure 4.2: Reduced cutoff time, *single best* (+), *uniform* (×), *ppfolio-like* approach (∗), *aspeed* (■), *selection* (□), *oracle* (◯).

best algorithm (+) got worse with an increased cutoff time on the training data of *Random*. On the *Crafted* set (Figure 4.2b), the performance of *aspeed* was found to benefit from an increased cutoff time, but the improvement was small for a cutoff time longer than 2222 CPU seconds (4/9 of the maximal cutoff time). In contrast, the improvement of the *ppfolio-like* approach (∗) was small on *Crafted* and *Random*; and the performance of *aspeed*, *ppfolio-like* approach and *single best* fluctuated on the *Application* set (Figure 4.2c). All three approaches benefited from the maximal cutoff time (5000 CPU seconds); however, the benefit was small in comparison to *aspeed* with the fully reduced cutoff time (195 CPU seconds). We conjecture that in the case of *Crafted*, the easy instances are not representative for the harder instances in the test set, unlike in the case of *Random*, where all instances were randomly generated and of similar structure. Consequently, on sets like *Random*, easier instances can be used for the computation of a schedule, even if the resulting schedule is ultimately applied to (and evaluated on) harder instances.

In an additional experiment, we assessed the performance of *aspeed* in the context of preparing for a competition. *aspeed* was trained on instances of the 2009 SAT Competition with the SAT solvers *CryptoMiniSat*, *clasp* and *tnm*, which are the same solvers used by *ppfolio*, and evaluated on the instances of the 2011 SAT Competition; see Table 4.7. On the entire instance set, *aspeed* had a PAR10 of 21196, in contrast to the *single best* algorithm with 32457 (a factor

	Random	*Crafted*	*Application*	*Complete*
single best	23662	29906	16942	32457
aspeed	19061	24623	16942	21196

Table 4.7: PAR10 of *single best* and *aspeed*, trained on 2009 SAT Competition and evaluated on 2011 SAT Competition.

1.53 higher). Also, *aspeed* outperformed the *single best* algorithm on *Random* and *Crafted*, and it performed just as well as the *single best* algorithm on *Application*. This latter observation is due the fact that the performance of *CryptoMiniSat* dominated on the *Application* set, and hence, *aspeed* was unable to obtain improved performance on *Application*.

4.4.7 Comparison with *3S*

In our final experiment, we compared *aspeed* with the SAT solver *3S*, which uses an approach similar to *aspeed*, but combines a static algorithm schedule with algorithm selection based on instance features (see Section 4.5). Since only the sequential version of the solver *3S* is freely available but not the schedule building method, we could not train the models of *3S* on new benchmark sets. Therefore, we trained *aspeed* on the same training runtime measurements used by the authors of *3S* for training on the 2011 SAT Competition, namely the *3S-Set*. We note that training of *3S*, unlike *aspeed*, additionally requires a set of instance features. Using these versions of *aspeed* and *3S* trained on the same set of instances, we measured the runtime of both solvers (utilizing a single processor *SP* or multi-processor environment with four parallel threads *MP4*) on the instances of the 2011 SAT Competition with the same cutoff of 5000 CPU seconds as used in the competition.

	Random	*Crafted*	*Application*	*Complete*
3S	16415	23029	19817	18919
aspeed-SP	22095	22180	24579	22737
aspeed-4P	16380	20142	17164	17517

Table 4.8: PAR10 of *3S* and *aspeed*, trained on the training data of *3S* and evaluated on 2011 SAT Competition.

Table 4.8 shows the results based on the PAR10 of the runtime measurements. The results are similar to the comparison between *SATzilla* and *aspeed*. The single processor version of *aspeed*, *aspeed-SP*, outperformed *3S* on *Crafted* in the sequential case. This could indicate that the instance feature set, used by *SATzilla* and *3S*, does not sufficiently reflect the runtime behaviour of the individual algorithms on these types of instances. Furthermore, *aspeed* with four cores, *aspeed-4P*, performed better than *3S* on all three instance sets.

4.5 Related Work

Our work forms part of a long line of research that can be traced back to John Rice's seminal work on algorithm selection (1976) on one side, and to work by Huberman, Lukos, and Hogg

(1997) on parallel algorithm portfolios on the other side.

Most recent work on algorithm selection is focused on mapping problem instances to a given set of algorithms, where the algorithm to be run on a given problem instance i is typically determined based on a set of (cheaply computed) features of i. This is the setting considered prominently in (Rice, 1976), as well as by the work on SATzilla, which makes use of regression-based models of running time (Xu, Hoos, & Leyton-Brown, 2007; Xu et al., 2008); work on the use of decision trees and case-base reasoning for selecting bid evaluation algorithms in combinatorial auctions (Guerri & Milano, 2004; Gebruers, Guerri, Hnich, & Milano, 2004); and work on various machine learning techniques for selecting algorithms for finding maximum probable explanations in Bayes nets in real time (Guo & Hsu, 2004). All these approaches are similar to ours in that they exploit complementary strengths of a set of solvers for a given problem; however, unlike these per-instance algorithm selection methods, *aspeed* selects and schedules solvers to optimize performance on a set of problem instances, and therefore does not require instance features.

It may be noted that the use of pre-solvers in *SATzilla*, that is, solvers that are run feature-extraction and feature-based solver selection, bears some resemblance to the sequential solver schedules computed by *aspeed*; however, *SATzilla* considers only up to 2 pre-solvers, which are determined based on expert knowledge (in earlier versions of SATzilla) or by exhaustive search, along with the time they are run for.

CPhydra is a portfolio-based procedure for solving constraint programming problems that is based on case-based reasoning for solver selection and a simple complete search procedure for sequential solver scheduling (O'Mahony et al., 2008). Like the previously mentioned approaches, and unlike *aspeed*, it requires instance features for solver selection, and, according to its authors, is limited to a low number of solvers (in their work, five). Like the simplest variant of *aspeed*, the solver scheduling in *CPhydra* aims to maximize the number of given problem instances solved within a given time budget.

Early work on parallel algorithm portfolios highlights the potential for performance improvements, but does not provide automated procedures for selecting the solvers to be run in parallel from a larger base set (Huberman et al., 1997; Gomes & Selman, 2001). *ppfolio*, which demonstrated impressive performance at the 2011 SAT Competition, is a simple procedure that runs between 3 and 5 SAT solvers concurrently (and, depending on the number of processors or cores available, potentially in parallel) on a given SAT instance. The component solvers have been chosen manually based on performance on past competition instances, and they are all run for the same amount of time. Unlike *ppfolio*, our approach automatically selects solvers to minimize the number of timeouts or total running time on given training instances using a powerful ASP solver and can, at least in principle, work with much larger numbers of solvers. Furthermore, unlike *ppfolio*, *aspeed* can allot variable amounts of time to each solver to be run as part of a sequential schedule.

Concurrently with our work presented here, Yun and Epstein (2012) developed an approach that builds sequential and parallel solver schedules using case-based reasoning in combination with a greedy construction procedure. Their RSR-WG procedure combines fundamental aspects of *CPhydra* (O'Mahony et al., 2008) and GASS (Streeter, Golovin, & Smith, 2007); unlike *aspeed*, it relies on instance features. RSR-WG uses a relatively simple greedy heuristic to optimize the number of problem instances solved within a given time budget by the parallel solver schedule to be constructed; our use of an ASP encoding, on the other hand, offers considerably

more flexibility in formulating the optimization problem to be solved, and our use of powerful, general-purpose ASP solvers can at least in principle find better schedules. Our approach also goes beyond RSR-WG in that it permits the optimization of parallel schedules for runtime.

Gagliolo and Schmidhuber consider a different setting, in which a set of algorithms is run in parallel, with dynamically adjusted timeshares (2006). They use a multi-armed bandit solver to allocate timeshares to solvers and present results using two algorithms for SAT and winner determination in combinatorial auctions, respectively. Their technique is interesting, but considerably more complex than *aspeed*; while the results for the limited scenarios they studied are promising, so far, there is no indication that it would achieve state-of-the-art performance in standardized settings like the SAT competitions.

For AI planning, Helmert et al. implemented the portfolio solver *Fast Downward Stone Soup* (Helmert et al., 2011; Seipp et al., 2012) which statically schedules planners. In contrast to *aspeed*, *Fast Downward Stone Soup*s computes time slices using a greedy hill climbing algorithm that optimizes a special planning performance metric, and the solvers are aligned heuristically. The results reported by Seipp et al. (2012) showed that an uniform schedule achieved performance superior to that of *Fast Downward Stone Soup*. Considering our results about uniform schedules and schedules computed by *aspeed*, we have reason to believe that the schedules optimized by *aspeed* could also achieve performance improvements on AI planning problems.

Perhaps most closely related to our approach is the recent work of Kadioglu et al. on algorithm selection and scheduling (Kadioglu et al., 2011), namely *3S*. They study pure algorithm selection and various scheduling procedures based on mixed integer programming techniques. Unlike *aspeed*, their more sophisticated procedures rely on instance features for nearest-neighbour-based solver selection, based on the (unproven) assumption that any given solver shows similar performance on instances with similar features (Kadioglu et al., 2010). (We note that solver performance is known to vary substantially over sets of artificially created, 'uniform random' SAT and CSP instances that are identical in terms of cheaply computable syntactic features, suggesting that this assumption may in fact not hold.) The most recent version of *3S* (Malitsky et al., 2012) also supports the computation of parallel schedules but is unfortunately not available publicly or for research purposes. We focussed deliberately on a simpler setting than their best-performing semi-static scheduling approach in that we do not use per-instance algorithm selection, yet still obtain excellent performance. Furthermore, *3S* only optimizes the number of timeouts whereas *aspeed* also optimizes the solver alignment to improve the runtime.

4.6 Conclusion

In this work, we demonstrated how ASP formulations and a powerful ASP solver (*clasp*) can be used to compute sequential and parallel algorithm schedules. In principle, a similar approach could be pursued using CP or ILP as done within *3S* (Kadioglu et al., 2011). However, as we have shown in this work, ASP appears to be a good choice, since it allows for a compact and flexible encoding of the specification, for instance, by supporting true multi-objective optimization, and can be applied to effectively solve the problem for many domains.

Compared to earlier model-free and model-based approaches (*ppfolio* and *SATzilla*, respec-

tively), our new procedure, *aspeed*, performs very well on ASP, CSP, MaxSAT, QBF and SAT – five widely studied problems for which substantial and sustained effort is being expended in the design and implementation of high-performance solvers. In the case of SAT, there is no single dominant algorithm, and portfolio-based approaches leverage the complementary strength of different state-of-the-art algorithms. For ASP, a situation exists with respect to different configurations of a single solver, *clasp*. This latter case is interesting, because we essentially use *clasp* to optimize itself. While, in principle, the kind of schedules we construct over various configurations of *clasp* could even be used *within aspeed* instead of plain *clasp*, we have not yet investigated the efficacy of this approach.

Our open-source reference implementation of *aspeed* is available online. We expect *aspeed* to work particularly well in situations where various different kinds of problem instances have to be solved (for example, competitions) or where single good (or even dominant) algorithms or algorithm configurations are unknown (for example, new applications). Our approach leverages the power of multi-core and multi-processor computing environments and, because of its use of easily modifiable and extensible ASP encodings, can in principle be readily modified to accommodate different constraints on and optimization criteria for the schedules to be constructed. Unlike most other portfolio-based approaches, *aspeed* does not require instance features and can therefore be applied more easily to new problems.

Because, like various other approaches, *aspeed* is based on minimization of timeouts, it is currently only applicable in situations where some instances cannot be solved within the time budget under consideration (this setting prominently arises in many solver competitions). In future work, we intend to investigate strategies that automatically reduce the time budget if too few timeouts are observed on training data; we are also interested in the development of better techniques for directly minimizing runtime.

In situations where there is an algorithm or configuration that dominates all others across the instance set under consideration, portfolio-based approaches are generally not effective (with the exception of performing multiple independent runs of a randomized algorithm). The degree to which performance advantages can be obtained through the use of portfolio-based approaches, and in particular *aspeed*, depends on the degree to which there is complementarity between different algorithms or configurations, and it would be interesting to investigate this dependence quantitatively, possibly based on recently proposed formal definitions of instance set homogeneity (Schneider & Hoos, 2012). Alternatively, if a dominant algorithm configuration is expected to exist but is unknown, such a configuration could be found using an algorithm configurator, for instance *ParamILS* (Hutter, Hoos, & Stützle, 2007; Hutter et al., 2009), *GGA* (Ansótegui et al., 2009), *F-Race* (López-Ibáñez et al., 2011) or *SMAC* (Hutter et al., 2011a). Furthermore, automatic methods, like *hydra* (Xu et al., 2010) and *isac* (Kadioglu et al., 2010), construct automatically complementary portfolios of algorithm configurations with the help of algorithm configurators which could be also combined with *aspeed* to further increase its performance.

5 Advances in Algorithm Selection for Answer Set Programming

Answer Set Programming (ASP; (Baral, 2003)) has become a popular approach to declarative problem solving. This is mainly due its appealing combination of a rich and simple modeling language with high performance solving technology. ASP decouples problem specifications from solving algorithms; however, modern ASP solvers are known to be sensitive to search configurations – a phenomenon that is common to advanced Boolean constraint processing techniques. To avoid the necessity of manual solver configuration, a substantial amount of research was thus devoted to automated algorithm configuration and selection approaches, as we detail in Section 5.1; in ASP, we find work by Gebser et al.(2011), Hoos et al.(2012), Maratea, Pulina, and Ricca(2012), Silverthorn, Lierler, and Schneider(2012) and Maratea, Pulina, and Ricca(2013), and in particular the two portfolio-based systems *claspfolio* (Gebser et al., 2011) and *ME-ASP* (Maratea et al., 2013). The idea of such portfolio-based systems is to train classifiers on features of benchmark instances in order to predict the putatively best solver from a given solver portfolio. The portfolio of solvers used in this approach may consist of distinct configurations of the same solver or contain different solvers.

In what follows, we describe the new portfolio-based ASP system *claspfolio*, whose earlier version 1.0 won first, second, and third places at various ASP competitions. Version 0.8 of *claspfolio* was briefly described in a short paper by Gebser et al. (2011) and is conceptually identical to the first stable release of version 1.0. The key design features of this prototype were (i) feature generation using a light-weight version of the ASP solver *clasp*, the original *claspre* system, (ii) performance estimation of portfolio solvers via support vector regression, and (iii) a portfolio consisting of different *clasp* configurations only. In contrast to this rigid original design, the new version 2 of *claspfolio* provides a modular and open architecture (Section 5.2) that allows for integrating several different approaches and techniques. This includes (i) different feature generators, (ii) different approaches to solver selection, (iii) variable solver portfolios, as well as (iv) solver-schedule-based pre-solving techniques. The default setting of *claspfolio* 2 relies on an advanced version of *claspre* (Section 5.3), a light-weight version of *clasp* that produces statistics based on which numerous static and dynamic instance features are generated.

The flexible and open design of *claspfolio* 2 is a distinguishing factor even beyond ASP. As such, it provides a unique framework for comparing and combining existing approaches and techniques in a uniform setting. We take advantage of this and conduct an extensive experimental study comparing the influence of different options regarding (i), (ii), and (iii). In addition to gaining insights into the impact of the various approaches and techniques, we identify distinguished options showing substantial performance gains not only over *clasp*'s default configuration but moreover over manually tuned configurations of *clasp*. *claspfolio* 2 is 19-51% faster than the best known static *clasp* configuration and also 14-37% faster than *claspfolio* 1.0, as shown in Table 5.8 at the end of the paper. To facilitate reproducibility of our results and to

promote the use of high-performance ASP solving technology, we have made *claspfolio* 2 publicly available as open-source software at `http://potassco.sourceforge.net/#claspfolio`.

5.1 Related Work

Our work continues a long line of research that can be traced back to John Rice's seminal work on algorithm selection (Rice, 1976) on one side, and to work by Huberman et al. (1997) on parallel algorithm portfolios on the other side. Especially on SAT problems, automatic algorithm selectors have achieved impressive performance improvements in the last decade. *SATzilla* ((Xu et al., 2008, 2008, 2007; Xu, Hutter, Hoos, & Leyton-Brown, 2009, 2011)) predicted algorithm performance by means of ridge regression until 2009 and nowadays uses a pairwise voting scheme based on random forests; *isac* (Kadioglu et al., 2010) clusters instances in the instance feature space and uses a nearest neighbour approach on cluster centers for algorithm selection; *3S* (Kadioglu et al., 2011; Malitsky et al., 2013b) uses k-NN in the feature space and introduces pre-solving schedules computed by Integer Linear Programming and cost-sensitive clustering; *SNAPP* (Collautti, Malitsky, Mehta, & O'Sullivan, 2013) predicts algorithm performance based on instance features and chooses an algorithm based on the similarity of the predicted performances. All these systems are specialized on a single approach. They are highly efficient but do not provide a uniform setting, that is, different inputs and different performance metrics.

Apart from SAT, there exist several algorithm selectors for other problems. Following the original *claspfolio* of Gebser et al. (2011) approach, Maratea et al. (2012) presented *ME-ASP*, a multi-engine algorithm selector for ASP with an instance feature generator for syntactic features. Similarly, *AQME* (Pulina & Tacchella, 2007) is a multi-engine selector for QSAT. *CPhydra* (O'Mahony et al., 2008) selects a set of CSP solvers based on case-based reasoning and schedules them heuristically. *Fast Downward Stone Soup* (Seipp et al., 2012; Helmert et al., 2011) uses greedy hill climbing to find algorithm schedules for planning problems. *aspeed* (Hoos et al., 2015) also computes algorithm schedules, but takes advantage of the modeling and solving capabilities of ASP to find timeout-minimal schedules.

Related to our work on a more general level, Hutter, Xu, H.Hoos, and Leyton-Brown (2014b) gave an overview over runtime prediction techniques, which is also used in some algorithm selection approaches, for example, *SATzilla*'09. A comparison of different machine learning algorithms for algorithm selection was presented by Kotthoff, Gent, and Miguel (2012). Based on these results, Kotthoff (2013) introduced *LLAMA*, Leveraging Learning to Automatically Manage Algorithms, a flexible framework that provides functionality to train and assess the performance of different algorithm selection techniques.

5.2 Generalized Algorithm Selection Framework

The algorithm framework of *claspfolio* 2 combines the flexibility of *LLAMA* with additional state-of-the-art techniques and produces an executable algorithm selection solver. As such, it provides a unique framework for comparing and combining existing approaches and techniques in a uniform setting. Furthermore, the new design of *claspfolio* 2 follows the idea of Level 4 of *programming by optimisation* (Hoos, 2012): *"The software-development process is centered on the*

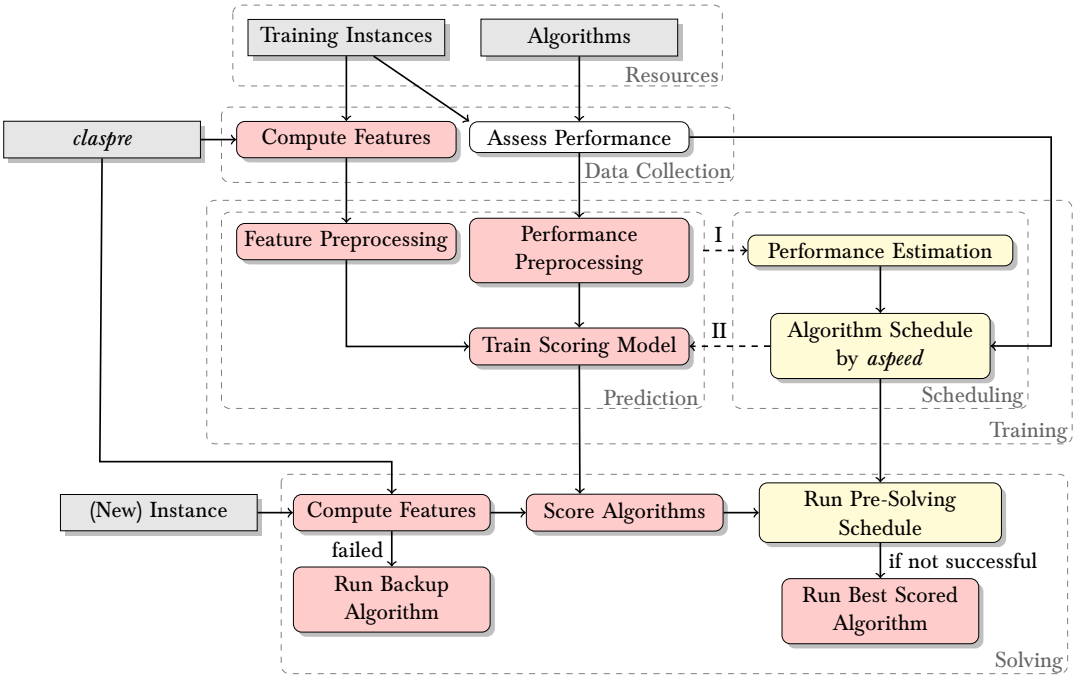


Figure 5.1: General workflow of *claspfolio* 2. Objects such as algorithms and instances are shown as rectangles, and activities are depicted as rectangles with rounded corners. Activities related to algorithm are tinted red and activities related to algorithm schedules yellow.

idea of providing design choices and alternatives in all parts of a project that might benefit from them; design choices that cannot be justified convincingly are not made prematurely."

A further distinguishing feature of the *claspfolio* 2 framework is the efficient and deep integration of an algorithm scheduling system, viz. *aspeed* (Hoos et al., 2015), into an algorithm selection framework to compute a static pre-solving schedule. *claspfolio* 2 uses *aspeed* to determine the running times used within pre-solving schedules. Thereby, it considers the estimated quality of the algorithm selector to determine the running time of the complete pre-solving schedule. This also allows us to integrate the pre-solving strategies of *SATzilla* and *3S*.

The general workflow underlying *claspfolio* 2 consists of collecting training data, learning a prediction model and training a pre-solving schedule; the portfolio-based ASP solver thus obtained solves a given problem instance with the pre-solving schedule and a solver selected by the prediction model. In what follows, we describe how this workflow is implemented efficiently in *claspfolio* 2; see Figure 5.1.

1. Resources. To train an algorithm selector, *training instances* and a portfolio of *algorithms* are required. Algorithm selection is based on the assumption that the given *training instances* are representative for the instances to be solved using the trained algorithm selection solver. In addition, a portfolio, that is, a set of *algorithms* with complementary strengths (for example, high-performance solvers used in a competition), provides the basis for algorithm selectors to efficiently solve a large variety of instances.

2. Data Collection. An algorithm selection task is defined based on the performance of all algorithms on all training instances (*Assess Performance*), instance features for each instance (*Compute Features*) and the costs for feature computation define an algorithm selection task.*claspfolio* 2 supports several feature generators, of which *claspre* is used by default.

3. Training. The training phase of *claspfolio* 2 makes use of two distinct components: *Prediction* and *Scheduling*. Both components can also be used separately in *claspfolio* 2.

The *Prediction* component of *claspfolio* 2 involves *feature pre-processing*, for example, feature normalization and feature selection, and *performance pre-processing*, for example, performance score transformation and algorithm filtering[1]. Based on the preprocessed data, a *scoring model* is learned, which maps the feature vector for a given problem instance to scores for all *algorithms* such that algorithms expected to perform well on the given instances are assigned better scores.

The *Scheduling* component of *claspfolio* 2 computes a timeout-minimal pre-solving schedule using *aspeed* (Hoos et al., 2015), where each algorithm gets a (potentially zero) time slice of the overall runtime budget available for solving a given problem instance. If the *prediction* component is not used, the schedule consists only of the given *algorithms*. If the *prediction* component is used, cross validation is used to obtain an unbiased estimate of the performance (*Performance Estimation*) of the *prediction* component (Arrow I). The resulting performance estimate of the prediction component is used as an additional simulated algorithm in the schedule generation process. All components of the schedule except the simulated one form the pre-solving schedule used in *claspfolio* 2. If the *prediction* performs well, the pre-solving schedule may be empty because the pre-solving schedule cannot perform better than a perfect predictor, that is, the selection of the best solver. In contrast, if *prediction* performs very poorly (for example, as a result of non-informative instance features), the simulated algorithm may be assigned a time slice of zero seconds and the prediction component is de facto ignored in the *solving* step.

Like *SATzilla* (Xu et al., 2008), *claspfolio* 2 allows to ignore instances solved by the pre-solving schedule (Arrow II) when learning the scoring model, such that the resulting model is focused on the harder instances not solved by the pre-solvers that are actually subject to algorithm selecting during the solving phase.

4. Solving a *(new) instance* starts with the computation of its features. If feature computation fails, for example, because it requires too much time, a *backup solver* is used to solve the instance. Otherwise, the scoring model is used to *score* each algorithm of the portfolio based on the computed feature vector. If the algorithm with the best score is part of the pre-solving schedule, it is removed from the schedule, because running the same algorithm twice does not increase the solving probability (when using deterministic algorithms like *clasp*). Next, the *pre-solving schedule* is executed.[2] If at the end of executing the pre-solving schedule, the instance has not been solved, the algorithm with the highest score is run for the remainder of the overall time budget.

[1] Algorithm filtering removes components of the portfolio given some strategy, for example, algorithms with a marginal contribution on virtual best solver performance of 0 can be removed. In (Xu et al., 2008), this is called solver subset selection and in (Maratea et al., 2012), solver selection.

[2] Unlike this, *SATzilla* runs the pre-solving schedule first and then computes the instance features, because the feature computation can be costly in SAT and the pre-solving schedule can solve the instance without incurring this cost. However, this does not permit removal of the selected solver from the pre-solving schedule.

Is Tight?	# Rules	% Integrity Constraints
# Problem Variables	#, % Normal Rules	# Equivalences
# Free problem Variables	#, % Cardinality Rules	#, % Atom-Atom Equivalences
# Assigned problem Variable	#, % Choice Rules	#, % Body-Body Equivalences
# Constraints	#, % Weight Rules	#, % Other Equivalences
# Constraints / #Variables	% Negative body Rules	#, % Binary Constraints
# Created Bodies	% Positive body Rules	#, % Ternary Constraints
# Program Atoms	% Unary Rules	#, % Other Constraints
# SCCs	% Binary Rules	
# Nodes in positive BADG	% Ternary Rules	

Table 5.1: 38 static features computed by *claspre* (# = number, % = fraction, SCCs = Strongly Connected Components, BADG = Body-Atom-Dependency Graph)

# Choices	#, % Literals conflict nogoods	Longest backjump (bj)
# Conflicts / #Choices	#, % Literals loop nogoods	#, ∅ Skipped levels while bj
∅ conflict level	#, % Removed nogoods	running average Conflict level
∅ LBD level	#, % Learnt binary nogoods	running average LBD level
#, % Learnt conflict nogoods	#, % Learnt ternary nogoods	
#, % Learnt loop nogoods	#, % Learnt other nogoods	

Table 5.2: 25 dynamic features computed (at each restart) by *claspre* (# = number, % = fraction, ∅ = average, LBD = Literal Blocking Distance)

5.3 *claspre*: Instance Features for ASP

The entire concept of algorithm selection is based on instance features which characterize benchmark instances and allow for predicting the putatively best solver from a given portfolio. These instance features should be cheap-to-compute to save as much time as possible for the actual solving process, but should also provide sufficient information to distinguish between (classes of) instances for which different solvers or solver configurations work best.

For feature generation, *claspfolio* 2 uses *claspre* in its default configuration. *claspre* is a light-weight version of *clasp* (Gebser et al., 2011) that extracts instance features of ground ASP instances in *smodels* format (Syrjänen, 2001), using *clasp*'s internal statistics. The features determined by *claspre* can be grouped into static and dynamic ones. The former are listed in Table 5.1 and include 38 properties, such as number of constraints. Beyond that, *claspre* performs a limited amount of search to collect dynamic information about solving characteristics. These dynamic features are computed after each restart of the search process, where restarts are performed after a fixed number of conflicts. Thereby, 25 dynamic features (Table 5.2) are extracted after each restart, such as the average number of conflict levels skipped while back-jumping.

The number of restarts performed is a parameter of *claspre*. More restarts lead to longer feature vectors that may contain more information. The number of restarts and number of conflicts between restarts determine the time used by *claspre* for feature computation We note that the pre-processing and search performed by *claspre* can actually solve a given ASP instance. The probability of this happening increases with the length of the search performed within *claspre*; however, at the same time, long runs of *claspre* reduce the time available for running solvers from the portfolio.

5.4 Empirical Performance Analysis on ASP

As previously described, *claspfolio* 2's modular and open architecture (Section 5.2) allows for integrating several different approaches and techniques, including (i) different feature generators, (ii) different approaches to solver selection, as well as (iii) variable solver portfolios. Taking advantages of this flexibility, we conducted an extensive experimental study to assess the efficacy of the various choices on large and representative sets of ASP instances.

Training data of *claspfolio* 2 is stored in the algorithm selection data format of the Algorithm Selection Library (ASlib) developed by the *COSEAL Group*,[3] an international group of experts in the field of algorithm selection and configuration. Detailed experimental results and the source code of *claspfolio* 2 are available at `http://www.cs.uni-potsdam.de/claspfolio`. Our empirical analysis makes use of commonly used techniques from statistics and machine learning (see, for example, (Bishop, 2007)).

5.4.1 Setup

All our experiments were performed on a computer cluster with dual Intel Xeon E5520 quad-core processors (2.26 GHz, 8192 KB cache) and 48 GB RAM per node, running Scientific Linux (2.6.18-308.4.1.el5). Each algorithm run was limited to a runtime cutoff of 600 CPU seconds and to a memory cutoff of 6 GB. Furthermore, we used permutation tests with 100 000 permutations and significance level $\alpha = 0.05$ to our performance metrics, the (0/1) timeout scores, the PAR10 scores and the PAR1 scores,[4] to asses the statistical significance of observed performance differences.

5.4.2 Instance Sets

We used all instances submitted to the 2013 ASP Competition in the `NP` category that could be grounded with *gringo* (3.0.5) within 600 CPU seconds and 6 GB memory. The resulting instance set consists of 2214 instances from 17 problem classes; we call it *Comp-13-Set*. As an even more heterogeneous instance set, we used the ASP *Potassco-Set* introduced by Hoos et al. (2013); it consists of 2589 instances from 105 problem classes and includes instances from the ASP competitions organized in 2007 (SLparse track), 2009 (with the encodings of the Potassco group) and 2011 (decision NP-problems from the system track), as well as several instances from

[3] `https://code.google.com/p/coseal`

[4] PARX is the penalized average runtime penalizing timeouts by X times the runtime cutoff.

the ASP benchmark collection platform *asparagus*.[5] All instances were grounded with *gringo*, and the grounding time was not counted towards solving the instances.

Each instance set was randomly split into equally sized, disjoint training and test set; only the training sets were used in the process of building algorithm portfolios. The resulting *claspfolio* 2 solvers were evaluated on the hold-out test sets. We also used the training instances to determine the best *claspfolio* 2 configuration (Subsection 5.3). To assess the performance of *claspfolio* 2 (Subsection 5.6), we used a 10-fold cross validation on the test set. Notice that we cannot use the training set for *claspfolio* 2 to obtain an unbiased learned model, because the algorithm portfolios have an optimistic performance estimation on the training set on which they were build.

5.4.3 Building Algorithm Portfolios

In addition to a set of training instances, a portfolio (that is, a set) of algorithms is required to construct a portfolio solver. *claspfolio* 2 can handle portfolios containing different solvers as well as different configurations of a given solver, all of which are viewed as individual ASP solvers. We investigated the following portfolios of ASP solvers:

- *Expert-portfolio* of four *clasp* (2.1.3) configurations designed by Benjamin Kaufmann (configurations: *frumpy* (default), *jumpy*, *handy* and *crafty*)
- *SOTA-portfolio* (Maratea et al., 2012): non-portfolio solvers participating in the 2013 ASP Competition[6] and in addition, the well-established solvers *cmodels* and *smodels*; in detail: *clasp* (Gebser et al., 2011), *cmodels* (Giunchiglia et al., 2006), *lp2bv* (Nguyen, Janhunen, & Niemelä, 2013), *lp2mip* (Liu, Janhunen, & Niemelä, 2012), *lp2sat* (Janhunen, 2006), *smodels* (Simons et al., 2002), and *wasp* (Alviano, Dodaro, Faber, Leone, & Ricca, 2013)
- *Hydra-like-portfolio* (Xu et al., 2010, 2011) of *clasp* (2.1.3) configurations
- *ISAC-like-portfolio* (Kadioglu et al., 2010) of *clasp* (2.1.3) configurations

Expert-portfolio and *SOTA-portfolio* are portfolios manually constructed by experts. In contrast, *hydra* and *isac* are automatic methods for constructing portfolios using algorithm configurators, for example, *ParamILS* (Hutter et al., 2007), *GGA* (Ansótegui et al., 2009) or *SMAC* (Hutter et al., 2011a). They generate a portfolio of configurations of a given solver by determining configurations that complement each other well on a given set of training instances, with the goal of optimizing the performance of the portfolio under the idealized assumption of perfect selection; this performance is also called the virtual best solver (vbs) or oracle performance of the portfolio.

An implementation of *hydra* that can be applied to solvers for arbitrary problems has not yet been published by (Xu et al., 2010); therefore, we have implemented our own version of *hydra* (in consultation with the authors), which we refer to as *Hydra-like-portfolio* in the following. Also, since the only published version of *isac* (2.0) does not include algorithm configuration, we reimplemented the part of *isac* responsible for portfolio generation, dubbed *ISAC-like-portfolio*. In contrast to the original *isac*, which performs g-means clustering, *ISAC-like-portfolio* uses

[5] `http://asparagus.cs.uni-potsdam.de`

[6] *IDP3* was removed from the portfolio because it was strongly dominated by all other solvers.

	Comp-13-Set			*Potassco-Set*		
	#TOs	PAR10	PAR1	#TOs	PAR10	PAR1
Expert-portfolio	360	2169	255	100	491	74
SOTA-portfolio	335	1866	231	111	538	75
Hydra-like-portfolio	326	1798	207	**82**	**400**	**58**
ISAC-like-portfolio	**313**	**1724**	**196**	99	476	63

Table 5.3: Virtual best solver (VBS) performance of portfolio building approaches on test sets. Results shown in boldface were statistically significantly better than all others within the respective column (according to a permutation test with 100 000 permutations and $\alpha = 0.05$).

k-means clustering, where the number of clusters is determined by using cross-validation to optimize the scoring function of the k-means procedure (following Hoos et al. (2013)).

Using this approach, *ISAC-like-portfolio* found 15 clusters for *Comp-13-Set* and 11 clusters for *Potassco-Set*, inducing 15 and 11 configuration tasks, respectively. To obtain a fair comparison, we allocated the same time budget to *Hydra-like-portfolio* and allowed it to perform 15 and 11 iterations, respectively (each consisting of one configuration task). The configuration process performed by *SMAC* (2.06.01; (Hutter et al., 2011a)) on each cluster and in each *hydra* iteration, respectively, was allocated 120 000 CPU seconds, that is, 200 times the target algorithm cutoff time, and 10 independent repetitions, from which the result with the best PAR10 score on the given training set was selected. *SMAC* optimized PAR10.

Table 5.3 shows the performance of the virtual best solvers (that is, the performance of a perfect algorithm selector) for the different considered portfolios. Interestingly, the results differ qualitatively between two benchmark sets. While *SOTA-portfolio* performs better than *Expert-portfolio* on *Comp-13-Set*, *Expert-portfolio* is better on *Potassco-Set*. Furthermore, while for both sets, the automatic generation methods found better performing portfolios than the the manual selected methods, on the *Comp-13-Set*, *ISAC-like-portfolio* produced a better results than *Hydra-like-portfolio*, and the opposite holds for *Potassco-Set*. Furthermore, unlike conjectured by Maratea et al. (2012), a set of configurations of the same, highly parameterized solver (*Expert-portfolio*, *ISAC-like-portfolio* and *Hydra-like-portfolio*) generally did not yield worse performance than a mixed portfolio, such as *SOTA-portfolio*.

While we gave *hydra* the same time budget as *isac* to find portfolios, the components added by *Hydra-like-portfolio* in its final three iterations decreased the number of timeouts only by one on our training and test sets. Following Xu et al. (2010), *hydra* would be terminated when the performance does not improve on the training set after an iteration. Hence, *Hydra-like-portfolio* not only produced a better portfolio on *Potassco-Set* than *isac*, but also does so using less configuration time than *isac*.

5.4.4 Feature Sets

In addition to the *claspre* feature set presented in Section 5.3, we considered a set of ASP features introduced by Maratea et al. (2013) that is focussed on very efficiently computable syntactic features, such as number of variables. The published version of their feature generator supports

	Comp-13-Set					*Potassco-Set*				
	Min	$Q_{0.25}$	Median	$Q_{0.75}$	%TOs	Min	$Q_{0.25}$	Median	$Q_{0.75}$	%TOs
claspre(s)	0.04	1.43	1.72	8.83	16.2	0.13	0.91	1.38	1.72	1.0
claspre(s+d)	0.07	1.36	1.72	13.94	16.2	0.18	0.87	1.48	1.81	1.1
ME-ASP	0.04	1.18	1.97	15.97	3.2	0.06	0.83	1.10	1.79	0.1
lp2sat	0.08	24.88	484.85	600	49.4	0.04	3.81	21.82	91.13	14.6

Table 5.4: Time required for computing the features of a single ASP instance in CPU seconds, with a 600 seconds runtime cutoff. We report minimum (Min), 25% quartile ($Q_{0.25}$), median and 75% quartile ($Q_{0.75}$) of the distribution over the respective instance set, as well as the percentage of timeouts (%TOs).

only the ASPCore 1.0 (Calimeri, Ianni, & Ricca, 2011a) language of the 2011 ASP Competition. Our *Comp-13-Set* consists of instances of the 2013 ASP Competition in ASPCore 2.0, which introduced further language constructs. Therefore, we re-implemented this feature generator with the help of (Maratea et al., 2013) to be compatible with ASPCore 2.0.[7]

One of the most established and investigated feature generators for SAT is provided as part of *SATzilla* (Xu et al., 2008). ASP instances can be translated to SAT with techniques by Janhunen (2006), using his tool *lp2sat*. We use a combination of *lp2sat*[8] with the feature generator of *SATzilla* to generate a set of instance features for ASP instances; this is the first time, these features are studied in the context of ASP. Since the full set of *SATzilla* features is very expensive to compute and our SAT encodings can get quite large, we decided to only use the efficiently computable base features.

Table 5.4 shows the runtime statistics for *claspre* with static features, *claspre(s)*, *claspre* with static and dynamic features, *claspre(s+d)*, with 4 restarts and 32 conflicts between the restarts, the (re-implemented) feature generator of *ME-ASP* and the combination of *lp2sat* and *SATzilla*'s feature generator on our full benchmark sets (training + test instances). *claspre(s)* is only slightly faster than *claspre* with additional dynamic features, since its search was limited to 128 conflicts. To solve typical ASP instances, searches well beyond 100 000 conflicts are often required; nevertheless, *claspre(s)* solved 51 instances through pre-processing, and *claspre(s+d)* solved 123 instances on *Comp-13-Set*, 9 and 400 instances on *Potassco-Set*, respectively. The feature generation of *ME-ASP* was faster, but (unsurprisingly, considering the nature of these features) did not solve any instance. Because of the substantial overhead of generating translations from ASP to SAT, the combination of *lp2sat* and *SATzilla*'s feature generator turned out to be substantially slower than the other approaches and failed to compute the feature vectors of 1094 instances on *Comp-13-Set* and 377 instances on *Potassco-Set* within the given cutoff time.

5.4.5 Algorithm Selection Approaches

As previously mentioned, *claspfolio* 2 was explicitly designed to easily integrate several state-of-the-art algorithm selection approaches. This not only permits us to optimize the performance of *claspfolio* 2, but also to compare the considered algorithm selection approaches within a

[7] The new feature generator is implement in Python, whereas the original generator was implemented in C++, which induced an overhead of a factor 2 in terms of running time on average on ASPCore 1.0 instances from the 2011 ASP Competition.

[8] *lp2sat* was used as submitted at the 2013 ASP Competition.

	Approach	Feat. Norm.	Pre-Solver	Pre-Solver Time [sec]
aspeed	static schedule	none	$\leq \infty$	$\leq \infty$
claspfolio-1.0-like	SVR	z-score	0	0
ME-ASP-like	nearest neighbor	none	0	0
ISAC-like	k-means clustering	linear	0	0
3S-like	k-NN	linear	$\leq \infty$	$\leq$ cutoff/10
SATzilla'09-like	ridge regression	z-score	≤ 2	≤ 20
SATzilla'11-like	voting with random forest	z-score	≤ 3	≤ 30

Table 5.5: Excerpt of algorithm selection mechanism supported by *claspfolio* 2.

controlled environment. Although our re-implementations may not reproduce the original implementations in all details (something that would be difficult to achieve, considering that sources are not available for some published approaches), they provide the only freely available, open-source implementations of some of these systems and thus provide a basis for further analysis and improvements.[9]

Table 5.5 gives an overview of the approaches available within *claspfolio* 2. These differ with respect to (i) the algorithm selection method, (ii) the feature normalization technique, (iii) the maximal number of pre-solvers used and (iv) the maximal running time allocated to the pre-solving schedule. In all cases, the pre-solving schedules were computed by *aspeed*, and hyperparameters of the machine learning techniques were set using grid search on training data.

5.4.6 Results

We have assessed the performance of *claspfolio* 2 on all 112 combinations of our 4 feature sets, 4 portfolios and 7 algorithm selection approaches, using a cross validation on both test sets. To study the effect of each design choice, we collected statistics over the distribution of results by keeping one choice fixed and varying all remaining components; the results are shown in Table 5.6. The top part of the table shows results obtained for using each of the feature sets, in terms of average PAR10 performance, standard deviation in PAR10 performance and best PAR10 performance over all 28 combinations of portfolios and selection approaches. The subsequent parts of Table 5.6 show analogous results for different portfolios and selection approaches.

On average, the best feature set was *claspre(s)* (the static *claspre* features) on *Comp-13-Set*, followed by *claspre(s+d)* (the static + dynamic *claspre* features), the feature sets of *ME-ASP* and *lp2sat*. However, the best *claspfolio* 2 configuration on *Comp-13-Set* used *ME-ASP*. The fact that *claspre(s+d)* gave worse results than *claspre(s)*, although the former is superset of the latter, indicates that not all features were useful and that feature selection should be used to identify a subset of features with highest information content. On *Potassco-Set*, the best average performance and the best performance of any *claspfolio* 2 configuration was consistently obtained by

[9] As with *hydra* and *isac* above, published and trainable, general-purpose implementations of *3S* and *ME-ASP* are not available.

Impact of feature set				
	Comp-13-Set		*Potassco-Set*	
	$\mu_{PAR10} \pm \sigma_{PAR10}$	$\min_{PAR10}$	$\mu_{PAR10} \pm \sigma_{PAR10}$	$\min_{PAR10}$
claspre(s)	**2116.3** $\pm$ 128.7	1927.0	638.9 $\pm$ 81.1	490.6
claspre(s+d)	2127.6 $\pm$ 122.6	1931.3	**630.8** $\pm$ 78.1	**480.0**
ME-ASP	2138.4 $\pm$ 127.7	**1919.4**	661.0 $\pm$ 108.8	486.0
lp2sat	2240.3 $\pm$ 81.3	2056.9	688.3 $\pm$ 45.6	610.3
Impact of portfolio				
	Comp-13-Set		*Potassco-Set*	
	$\mu_{PAR10} \pm \sigma_{PAR10}$	$\min_{PAR10}$	$\mu_{PAR10} \pm \sigma_{PAR10}$	$\min_{PAR10}$
Expert-portfolio	2251.8 $\pm$ 55.0	2165.0	679.1 $\pm$ 47.7	621.6
SOTA-portfolio	2172.4 $\pm$ 60.6	2072.9	691.9 $\pm$ 55.3	614.7
Hydra-like-portfolio	2141.5 $\pm$ 160.4	1943.7	**609.6** $\pm$ 103.5	**480.0**
ISAC-like-portfolio	**2056.9** $\pm$ 111.3	**1919.4**	638.3 $\pm$ 90.9	526.7
Impact of selection mechanism				
	Comp-13-Set		*Potassco-Set*	
	$\mu_{PAR10} \pm \sigma_{PAR10}$	$\min_{PAR10}$	$\mu_{PAR10} \pm \sigma_{PAR10}$	$\min_{PAR10}$
aspeed	2292.8 $\pm$ 66.1	2222.0	731.2 $\pm$ 40.8	672.6
claspfolio-1.0-like	2152.7 $\pm$ 108.0	1978.6	650.3 $\pm$ 58.3	519.3
ME-ASP-like	2245.3 $\pm$ 77.3	2091.8	753.3 $\pm$ 76.7	656.8
ISAC-like	2100.1 $\pm$ 113.5	1939.5	608.4 $\pm$ 65.7	490.6
3S-like	2092.0 $\pm$ 109.2	1927.0	596.0 $\pm$ 57.6	489.1
SATzilla'09-like	2120.3 $\pm$ 99.4	1932.6	652.7 $\pm$ 48.2	544.0
SATzilla'11-like	**2086.4** $\pm$ 125.9	**1919.4**	**591.1** $\pm$ 62.5	**480.0**

Table 5.6: Statistics (μ = average, σ = standard deviation, min = minimum) of PAR10 performance over all combinations except for the one kept fixed to assess its impact.

using *claspre(s+d)*. We believe that the additional dynamic features are necessary to distinguish between the larger number of different problem classes in *Potassco-Set*.

The results on the impact of the portfolio of algorithms used as a basis for algorithm selection confirm our assumption that the best potential performance, that is, best VBS performance, is a good indicator of the actual performance achieved by a high-performance selection approach. On *Comp-13-Set*, *ISAC-like-portfolio* achieved the best performance, while on *Potassco-Set*, *Hydra-like-portfolio* yielded even better results. Furthermore, the portfolios obtained using the two automatic portfolio generation methods, *isac* and *hydra*, yielded better results than the manually created ones, *Expert-portfolio* and *SOTA-portfolio*.

As shown in the lower part of Table 5.6, the *SATzilla'11-like* approach performed best on both benchmark sets, followed closely by *3S-like* and *ISAC-like*. *SATzilla'09-like* and *claspfolio-1.0-like* showed similar, but weaker performance results, followed by the *ME-ASP-like* approach and the pure algorithm schedules of *aspeed*.

Overall, the best combination both on the training and test sets of *Comp-13-Set* was the *ME-ASP* features, *ISAC-like-portfolio* and *SATzilla'11-like* selection approach, and *claspre(s+d)* features, *Hydra-like-portfolio* and *SATzilla'11-like* selection approach for *Potassco-Set*.

Scenario	$\|I\|$	$\|U\|$	$\|A\|$	$\|F\|$
ASP-POTASSCO	1294	82	11	138
CSP-2010	2024	253	2	17
MAXSAT12-PMS	876	129	6	37
PREMARSHALLING-ASTAR-2013	527	0	4	16
QBF-2011	1368	314	5	46
SAT11-HAND	296	77	15	115
SAT11-INDU	300	47	18	115
SAT11-RAND	600	108	9	115
SAT12-ALL	1614	20	31	115
SAT12-HAND	767	229	31	115
SAT12-INDU	1167	209	31	115
SAT12-RAND	1362	322	31	115

Table 5.7: Overview of algorithm selection scenarios in Algorithm Selection Library with the number of instances $|I|$, number of unsolvable instances $|U|$ ($U \subset I$), number of algorithms $|A|$, and number of features $|F|$.

5.5 Empirical Performance Analysis on ASlib

The flexible framework of *claspfolio* 2 is not limited to ASP algorithms but it can also be applied to arbitrary algorithm selection scenarios. For this purpose, *claspfolio* 2 reads the format of the Algorithm Selection Library (ASlib[10]). We have assessed the selection approaches discussed in the previous section to investigate their strengths and weaknesses

5.5.1 Algorithm Selection Scenarios

ASlib contains a diverse set of algorithm scenarios from different applications. We note that the algorithm schedules implemented in *claspfolio* 2 via *aspeed* mainly optimize the number of timeouts which limits the application of *claspfolio* 2 to scenarios with runtime as performance type. 12 out of 14 scenarios have runtime as performance type, that is, *ASP-POTASSCO*, *CSP-2010*, *MAXSAT12-PMS*, *PREMARSHALLING-ASTAR-2013*, *QBF-2011*, *SAT11-INDU*, *SAT11-HAND*, *SAT11-RAND*, *SAT12-ALL*, *SAT12-INDU*, *SAT12-HAND* and *SAT12-RAND*. For a detailed description of these scenarios, we refer to Table 5.7.

5.5.2 Setup

Since the complete feature set of the SAT scenarios can generate a large amount of overhead, we use only the so-called 50 *base* features, which relate to the following feature computation steps: *Pre*, *Basic*, *KLB* and *CG*. Furthermore, we used the same algorithm selection approaches as described in Subsection 5.4.5. We use again a 10-fold cross validation, to unbiasedly assess the performance of *claspfolio* 2. Since *claspfolio* 2 cannot solve instances that are not solved by any selectable algorithm, we remove such instances from the test sets.

[10] `www.aslib.net`

5.5.3 Results

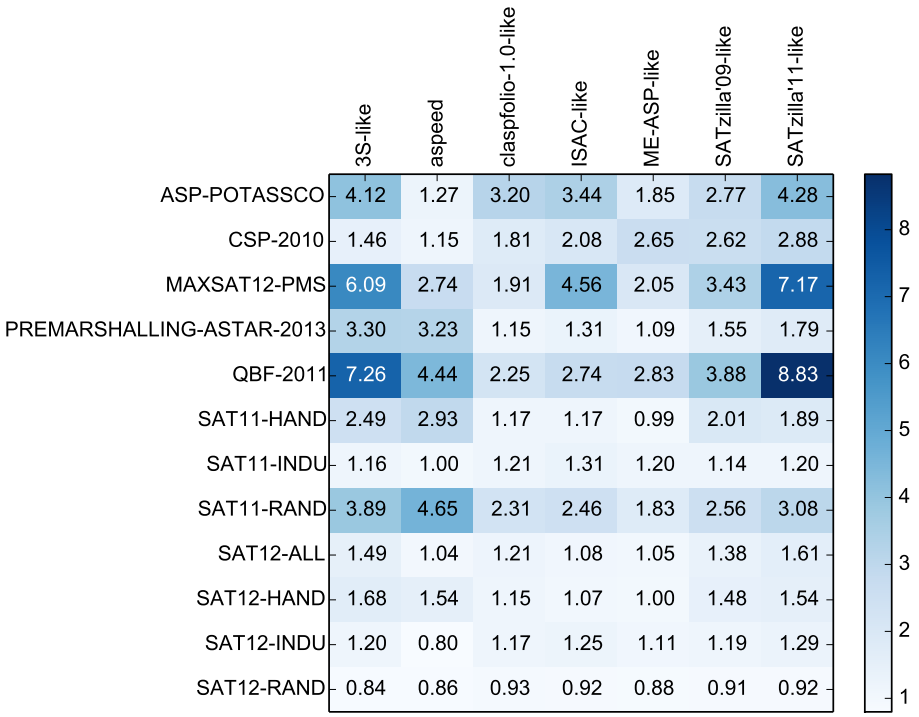


Figure 5.2: The color shading shows the factor by which the selection approach implemented in *claspfolio* 2 outperformed the *single best* on PAR10 without consideration of the unsolvable instances.

Figure 5.2 presents the performance of the different algorithm selection approaches in *claspfolio* 2 on the ASlib scenarios. The color shading and the values show the factor by which the selection approach outperformed the single best solver on PAR10. Detailed results - including number of timeouts and PAR1 - can be found in Appendix B.

5.5.4 Discussion

In contrast to our results for our ASP sets, *SATzilla'11-like* is not always the best approach. *SATzilla'11-like* is the best approach for 7 out of 12 scenarios, that is, *ASP-POTASSCO*, *CSP-2010*, *MAXSAT12-PMS*, *QBF-2011*, *SAT12-ALL*, *SAT12-INDU* and *SAT12-RAND*. Surprisingly, *SATzilla'11-like* is on the SAT scenarios not always the best approach even though it was developed in particular on this domain. We note that the original *SATzilla* implementation does not only consists of their selection approach, but its authors also collected the solvers in the algorithm portfolio, invented the instance features and uses further techniques to select a subset of features; this is more than we do in our experiments with *claspfolio* 2.

Furthermore, *aspeed* is twice the best solver on PAR10, that is, on *SAT11-HAND* and *SAT11-RAND*. However, it also solves the largest number of instances on *MAXSAT12-PMS* and *PREMARSHALLING-ASTAR-2013*. On these scenarios, it is not the best approach on PAR10 and

PAR1 because its algorithm schedule looses some time in unsuccessful algorithm runs before it runs the right solver that solves the instance.

Out of the scenarios, *PREMARSHALLING-ASTAR-2013* and *SAT12-RAND* are noticeable in particular. On *PREMARSHALLING-ASTAR-2013*, *3S-like* and *aspeed* perform very well. Both approaches use algorithm schedules with an unlimited number of constituent solvers for boosting their performance. *SATzilla'09-like* and *SATzilla'11-like* perform worse than these two because its pre-solving schedules are limited to 2 and 3 constituent solvers. Removing this restriction, increases also the performance of *SATzilla'-like* approaches. However, all approaches are not able to yield a performance matching a perfect algorithm selector. A perfect selector would get a speed up of 30.84 on *PREMARSHALLING-ASTAR-2013*. One possible reason could be that the instance features do not contain enough information for the selection approach.

On *SAT12-RAND*, all tested approaches have a worse performance than *single best*. This is consistent with the published results on ASlib. The authors of ASlib also only found one approach that was better than the *single best*, that is, a Random Forest regression. Furthermore, a look at the exploratory data analysis (EDA) at the ASlib online platform reveals that the *CG* features are not computed for 509 out of 1326 instances. Since *claspfolio* 2 does not impute missing features - in contrast to the ASlib reference approach - but uses a backup algorithm when the feature vector is incomplete, *claspfolio* 2 does no per-instance selection for these 509 instances. The PAR10 performance of *claspfolio* 2 with a Random Forest Regression without *CG* features is 3175. This is better than the *single best* with 3271 and the best result on ASlib with 3188.

So, we observed two things: (i) no selection approach dominates all other approaches and (ii) feature selection can be important to improve further the performance of *claspfolio* 2. However, a-priori it is not known which approach should be used of a given algorithm selection scenario. Therefore, one future step is to apply algorithm configuration to *claspfolio* 2 to get a well-performing scenario-specific configuration of *claspfolio* 2 (including feature selection).

5.6 Conclusion

Our new, modular *claspfolio* 2 ASP solver architecture comprises a diverse set of portfolio-based algorithm selection techniques, including feature extractors, manually and automatically constructed base algorithm portfolios, algorithm selection mechanisms and solver-schedule-based pre-solving techniques. As seen from the high-level overview of empirical performance results in Table 5.8, on standard, diverse and heterogeneous sets of ASP benchmarks, *claspfolio* 2 is substantially more robust than the default configuration of *clasp*, the manual tuned configuration of *clasp* of the 2013 ASP Competition, and than all other assessed individual solvers (including automatically configured clasp and other ASP solvers); in fact, its performance in terms of PAR10-score lies only about 20% and 15% above that of the best known oracle on *Potassco-Set* and *Comp-13-Set* benchmark sets, respectively. The reimplementation of *claspfolio* 1.0 in *claspfolio* 2, which had a similar performance in preliminary experiments than the original implementation, achieves also about $14-37\%$ higher PAR10-score than *claspfolio* 2. While the best configuration of *claspfolio* 2 varies between these two benchmark sets, the performance differences are relatively minor: on *Comp-13-Set*, the best configuration of *claspfolio* 2 for *Potassco-Set* – which we also chose as the default configuration for *claspfolio* 2 – achieves a

	Comp-13-Set			*Potassco-Set*		
	#TOs	PAR10	PAR1	*#TOs*	PAR10	PAR1
clasp (default)	577	3168	351	287	1347	176
clasp (ASP Comp 13)	421	2329	273	150	723	97
single best	414	2333	268	150	723	97
claspfolio 1.0	403	2237	269	134	658	99
claspfolio 2	**353**	**1960**	**237**	**97**	**480**	**75**
best known VBS	313	1724	196	82	400	58

Table 5.8: Comparison of two *clasp* configurations, the *single best* solver in all portfolios (cf. Subsection 5.3), *claspfolio* 1.0, the *claspfolio* 2 with *claspre(s+d)* features, *Hydra-like-portfolio* and *SATzilla'11-like* approach. The significantly best performances (except VBS) are shown in boldface (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$).

PAR10-score only about 2.1% lower than the best configuration for *Comp-13-Set*, and on *Potassco-Set*, its PAR10-score is about 9.6% higher. This configuration uses the *claspre(s+d)* feature set in combination with the *Hydra-like-portfolio* base algorithm portfolio construction approach and the *SATzilla'11-like* algorithm selection mechanism, but other feature sets, base algorithm portfolios and algorithm selection mechanisms also achieve very strong performance.

Also on a diverse set of algorithm selection scenarios from the Algorithm Selection Library, *claspfolio* 2 showed that it outperforms by a factor of up to 8.8 the single best solver on PAR10. However, the best selection approach varied between the scenarios.

In future work, we believe that further performance improvements could be achieved via automatic configuration of *claspfolio* 2. It exposes more than 40 performance relevant parameters. It is infeasible to manually configure such a large configuration space, so that automatic algorithm configurators, such as *SMAC*, should be used again.

6 Algorithm Selection of Parallel Portfolios

Current modern and highly efficient solvers are known to be performance sensitive to configurations of the search strategies. Apart from algorithm configuration, algorithm selection can be used to automatically construct robust algorithms, which are more effective than using a single algorithm (for an overview see, for example, Kotthoff et al. (2012)). Algorithm selection is based on the idea to select per-instance the putatively best algorithm or algorithm configuration. However, a learned mapping (for example, implemented with a machine learning model) from instance to an algorithm is not perfect in many cases, as we have seen in Section 5.5.

Since the increase of computational power is nowadays primarily achieved through additional parallel cores, the effectiveness of algorithms can not only be increased by selecting a single algorithm, but also by selecting a parallel portfolio of algorithms. For example, the SAT solver *CSCHpar* (Malitsky et al., 2013b, 2013a) won the open parallel track in the 2013 SAT Competition. The idea of *CSCHpar* is simple yet effective; *CSCHpar* always runs in parallel the parallel SAT solver *Plingeling* (Biere, 2010, 2011) with 4 threads, the sequential SAT Solver *CCASAT* (Cai et al., 2012) and three per-instance selected solvers. These per-instance solvers are selected by three models that are trained on application, hard-combinatorial and random SAT instances. However, *CSCHpar* is particularly designed for the SAT Competition with its 8 available cores and its three kinds of instances.

In this chapter, we focus on the fundamental problem of selecting a parallel portfolio in the following setting:

(i) one processing unit (for example, processor core) is exactly assigned to one algorithm (that is, no algorithm schedules are used);

(ii) there is no communication between the algorithms;

(iii) the size of the parallel portfolio can be adjusted arbitrarily, (that is, the overhead grows at most linearly with the size of the parallel portfolio);

(iv) we do not have special structural knowledge about the problem domain (for example, we do not know that SAT instances can be divided into three kinds).

CSCHpar does not fall into this setting, because it violates (ii) - (iv).

We present an approach to the problem of selecting a per-instance selected parallel portfolio, named *PASU*, which can be applied to arbitrary algorithm selection scenarios. It is built upon the assumption that selection of solvers can be associated with an uncertainty metric, that is, how sure is the selector that it selects the best algorithm for a given instance. If the uncertainty is zero and the prediction of the best solver is always correct, a parallel portfolio cannot improve the performance in comparison to perfect selection. However, if the selection of a solver is somehow uncertain, we try to select a parallel portfolio optimizing a distribution of predicted performance scores induced by an uncertainty measure. Using bootstrapping, our

approach can be applied to every algorithm selection approach that is based on performance predictions. Furthermore, to minimize the overhead generated by the selection phase, the approach is modelled in such a way that its complexity grows only linearly with the number of considered solvers and the size of the parallel portfolio.

The remaining chapter is structured as follows: First, work related to algorithm selectors for parallel portfolios is discussed in Section 6.1. Then, we present the theoretical foundations of our *PASU* approach in Section 6.2 and demonstrate its performance on a diverse and heterogeneous set of algorithm selection scenarios from the Algorithm Selection Library (ASlib[1]) in Section 6.3.

6.1 Related Work

Our work forms part of a long line of research that can be traced back to John Rice's seminal work on algorithm selection (1976) on one side, and to work by Huberman et al. (1997) on parallel algorithm portfolios on the other side. However, Huberman et al. did not provide automated procedures for selecting the solvers to be run in parallel from a larger algorithm set.

Gagliolo and Schmidhuber (2006) considered parallel portfolios with dynamically adjusted timeshares. For this, they used a multi-arm bandit model to periodically reallocate timeshares of solvers. However, so far they only showed that their approach worked for two algorithms in SAT and winner determination in combinatorial auctions.

As already discussed in the introduction, the cost-sensitive hierarchical clustering approach of *CSCH* (Malitsky et al., 2013b) was extended for the 2013 SAT Competition to select parallel portfolios in *CSCHpar* (Malitsky et al., 2013a). They used some constantly selected solvers (*Plingeling* with four threads and *CCASAT*) and three independently trained per-instance selection models. These models are trained on industrial, handcrafted and random SAT instances. Such an approach is only possible if several models can be learned for different sub-problems (for example, different tracks of the SAT Competition). Furthermore, the number of processes is not directly adjustable.

The extension of *3S* (Kadioglu et al., 2011), named *3Spar* (Malitsky et al., 2012), selects a parallel portfolio by using k-NN to find the k most similar instances in the feature space. With the help of Integer Linear Programming (ILP), *3Spar* constructs a per-instance parallel algorithm schedule based on training data of these k instances. A limitation of *3Spar* is that the complexity of the to be solved problem for every instance is *NP*-hard. It grows with the number of parallel processing units and number of available solvers.[2]

aspeed (see Chapter 4) solves a similar scheduling problem as *3Spar*, but does this during an off-line training phase. Therefore, *aspeed* does not generate overhead in the solving phase. Unlike *3Spar*, *aspeed* does not allow to include parallel solvers in the algorithm schedule and the algorithm schedule is static and not per-instance selected.

RSR-WG (Yun & Epstein, 2012) combines case-based-reasoning from *CPhydra* (O'Mahony et al., 2008) with a greedy construction of parallel portfolio schedules via GASS (Streeter et al., 2007) for CSP problems. The schedules are constructed per-instance, such that RSR-WG also relies on instance features. In the first step, a schedule is greedily constructed to maximize the

[1] `www.aslib.net`

[2] We note that *3Spar* is not available publicly or for research purposes.

number of solved instances and in the second step, the components of the schedule are spread over the available processing units. In this process, RSR-WG requires a method to determine similar training instances for a new given instance. Therefore, it is not applicable to other algorithm selection approaches, for examples, as used in *SATzilla*.

6.2 Algorithm Selection with Uncertainty

We start with the classical algorithm selection problem, as also stated for the Algorithm Selection Library (ASlib).

Per-instance algorithm selection problem Given a set I of instances of a problem and a probability distribution $\mathcal{D}$ over I, a set of algorithms A, and a performance metric $m : I \times A \to \mathbb{R}$ to be minimized, the objective in the *per-instance algorithm selection problem* is to find a mapping $\phi : I \to A$ that minimizes the expected performance $\mathbb{E}_{i \sim \mathcal{D}} m(i, \phi(i))$ we incur by running the selected algorithm $\phi(i)$ for instance i, where the expectation is taken with respect to instances $i \in I$ drawn from distribution $\mathcal{D}$.

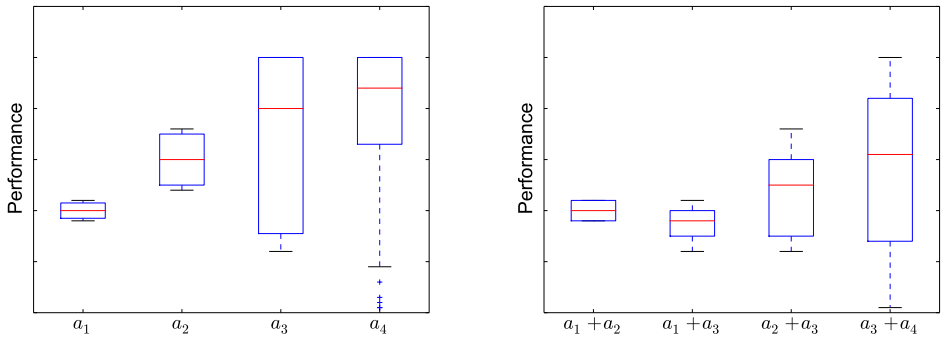
We extend this definition to the selection of parallel portfolios:

Per-instance parallel portfolio selection problem Given a set I of instances of a problem and a probability distribution $\mathcal{D}$ over I, a set of algorithms A, a set of processing units U and a performance metric $m : I \times A \to \mathbb{R}$ to be minimized, the objective in the *per-instance parallel portfolio selection problem* is to find mappings $\phi_u : I \to A$ for each processing unit $u \in U$ that minimize the minimal expected performance $\mathbb{E}_{i \sim \mathcal{D}} \min_{u \in U}(m(i, \phi_u(i)))$ we incur by running the selected algorithms $\{\phi_u(i)\}_{u \in \mathcal{U}}$ for instance i, where the expectation is taken with respect to instances $i \in I$ drawn from distribution $\mathcal{D}$.

In definition of the parallel selection problem, we assume that the selected algorithms are running in parallel with no communication between them. Therefore, the expected performance of such a portfolio is the performance of the best algorithm in the portfolio, that is, the optimal performance (e.g., minimal runtime) of all running $\phi_u(i)$ algorithms. In the following, we first present a straightforward approach for solving this problem, which directly extends the commonly used algorithm selection approach for parallel portfolio. Afterwards, we present our new approach, *PASU* which considers the estimated performance uncertainty of a selected algorithm.

In the following, we assume that we have a (potentially non-perfect) mapping $\phi : I \to A$ for the sequential algorithm selection problem and that the (machine learning) model $\mathcal{M}$ to implement this mapping ϕ will also return a performance estimation for each algorithm, for example, log-transformed runtime predictions, as used in *SATzilla*'09 (Xu et al., 2008).

Baseline. For a parallel portfolio of size k, a straightforward idea is to select the algorithms that have the best k predicted performance values. In our artificial example in Figure 6.1, the predicted performance of each algorithm obtained from the model $\mathcal{M}$ are marked in red. Algorithm a_1 has the best predicted performance, a_2 the second best and so on. Therefore, a portfolio of size 2 would consist of a_1 and a_2.



(a) predicted performance of algorithms

(b) predicted performance of parallel portfolios

Figure 6.1: Predicted performance (red line) with uncertainty (blue box with whiskers)

Algorithm 6: Training of *PASU*

Input : Algorithms $a \in A$, performance metric m, instances $i \in I$, training data $D_I = \langle (f(i), \langle m(i,a) \rangle_{a \in A})_{i \in I} \rangle$, subset size u, feature subset size v, ensemble size n

1 **for** $j := 1 \ldots n$ **do**
2 sample with bootstrapping a new training set D_I^j of size u from D_I
3 subsample v features from complete feature set and modify D_I^j accordingly
4 train model $\mathcal{M}_j$ on D_I^j that learns mapping $I \times A \to \mathbb{R}$
5 **return** $\langle (\mathcal{M}_j) \rangle_{j \in \{1 \ldots n\}}$

***PASU*: Parallel Algorithm Selection with Uncertainty.** Since we assume that the models are not perfect, there is uncertainty associated with the predictions obtained from each model (boxes and whiskers in Figure 6.1). We assume that we can also compute this uncertainty. In Figure 6.1, a_1 has the smallest predicted performance and hence the probability that prediction is incorrect is also small. In contrast, the predicted performance of a_4 is high and the uncertainty is also high; this means that the chance is high that a_4 can perform a lot better than expected. Taking into account the uncertainty of the prediction, the portfolio of size 2 with the best predicted performance would be a_1 and a_3, because a_1 has the best single predicted performance, and a_3 has a high uncertainty maybe inducing a better performance than a_1. The portfolio consisting of a_1 and a_2 – chosen in the baseline approach – has nearly the same predicted performance as a_1 alone, because the range of the uncertainty performances of a_2 is larger than the range of a_1. Therefore, we conclude that the expected performance does not provide enough information in all cases to select a parallel portfolio and we should also take into account the uncertainty. We call our new algorithm for **p**arallel **a**lgorithm **s**election with consideration of **u**ncertainty *PASU*.

How to Measure Uncertainty? Selection models, for example, as implemented in *claspfolio* (see Chapter 5) and *LLAMA* (Kotthoff, 2013), can return scores for each algorithms that are correlated with the predicted performance. For instance, regression models directly return a predicted performance score for each algorithm; or pairwise classification approaches (Xu et al., 2011) return votes for each algorithm. To determine the uncertainty of these predictions, in Algorithm 6, an ensemble of models is trained that returns a set of scores for each algorithm. Ensembles can be trained with bagging (subset sampling with bootstrapping) and feature subsampling to increase the diversity of the models. For example, we predict with two models ($n = 2$) that a_1 has performance 1.0 or 2.0 and a_2 has performance 1.5 and 2.5, respectively.

To run a parallel portfolio of algorithms, all portfolio components are started simultaneously. If one component was able to solve the instance, all other components are terminated. Therefore, the performance of a parallel algorithm portfolio is related to the component that solves an instance first. Since we have several predicted performance scores for each algorithm, the performance scores of a portfolio are the minima of all possible combinations of the algorithm scores. For the above example, all combinations of the performance scores are $(1.0, 1.5)$, $(1.0, 2.5)$, $(2.0, 1.5)$ and $(2.0, 2.5)$ and the scores of the portfolio, that is, the minimum of each tuple, are 1.0, 1.0, 1.5 and 2.0.

We note that this approach is based on the assumption that the scores of the algorithms are not related to each other. For example, the predicted runtimes of an algorithm is not related to the predicted runtime of another algorithm. However, the pairwise classification voting scores of *SATzilla*'11 (Xu et al., 2011) are related, that is, if an algorithm gets all votes to be best in comparison to all other algorithms, other algorithms cannot have the same number of votes.

How to Optimize with Uncertainty? To construct a parallel portfolio, we have to decide between several possible portfolios. Given the predicted score distribution of a portfolio, there are several ways to construct a parallel portfolio:

1. Minimize the expected performance of the portfolio; we note that this is not necessarily the same as using only one model and looking only at expected performances, as shown above in Figure 6.1
2. Minimize the upper part of the score distribution, that is, a pessimistic risk estimate to reduce the chance of poor decisions
3. Minimize the lower part of the score distribution, that is, an optimistic estimate to increase the chance to select a very well performing portfolio

In a general view, we can optimize a percentile q of the score distribution of a portfolio.

Greedy Portfolio Selection. Algorithm 7 shows how a portfolio can be built efficiently. Since the time complexity of selecting the best portfolio grows exponentially with the size of the portfolio, we use a greedy approach that selects a portfolio of size k in time $O(k \cdot |A|)$. In Line 1 and 2, the scores $\vec{s}_a$ for all algorithms $a \in A$ are predicted. We start with an empty portfolio in Line 3 and iterate as long as the portfolio has not reached the desired size k in Line 4. In Line 5 to 10, we compute the portfolio scores for all possible extended portfolios $P_{j-1}||a$. Since the cross product of all predicted scores in iteration j has size n^j, we approximate the score

Algorithm 7: Greedy Portfolio Selection in *PASU*

Input : Algorithms $a \in A$, instances $i \in I$, trained models $\langle(\mathcal{M}_j)\rangle_{j\in\{1\dots n\}}$, feature vector $\vec{f}$, portfolio size k, sample size r, percentile q

1 **forall the** $a \in A$ **do**
2 let $\vec{s}_a$ be the scores for algorithm a returned by $\langle(\mathcal{M}_j)\rangle_{j\in\{1\dots n\}}$ given $\vec{f}$
3 $P_0 := []$
4 **for** $j := 1 \dots k$ **do**
5 **forall the** $a \in A \backslash P_{j-1}$ **do**
6 $\vec{s}_{P_{j-1}||a} := \langle\rangle$ `// predicted scores of extended portfolio;`
7 **for** $l := 1 \dots r$ **do**
8 $\vec{s}^{(l)}_{P_{j-1}||a} := min(\langle(s'_{a^*})\rangle_{a^*\in P_{j-1}||a})$ where s'_{a^*} is a random element from $\vec{s}_{a^*}$, a^* is an algorithm in the extended portfolio $P_{j-1}||a$ and $\vec{s}^{(l)}_{P_{j-1}||a}$ is the l-th element of the vector $\vec{s}_{P_{j-1}||a}$
9 choose algorithm $\hat{a} \in A \backslash P_{j-1}$ minimizing the percentile q of $\vec{s}_{P_{j-1}||\hat{a}}$
10 $P_j := P_{j-1}||\hat{a}$
11 **return** P_k

distribution by sampling a subset of scores $\vec{s}_{P_{j-1}||a}$ (Lines 6 to 8). Then, we extend the portfolio with $\hat{a}$ which minimizes the percentile of the portfolio scores. In Line 11, we return the selected portfolio of size k.

6.3 Empirical Performance Analysis

We implemented our *PASU* approach within the flexible framework of *claspfolio* 2. As described, Algorithm 7 has a crucial parameter for the performance of *PASU*, that is, the percentile q of the score distribution to be optimised. We investigate the impact of this parameter on the performance here. Furthermore, we compare our *PASU* approach with the baseline approach and later on, with static parallel algorithm schedules from *aspeed* to show advantages and limitations of *PASU*.

6.3.1 Setup

Algorithm Selection Scenarios. We use algorithm selection scenarios from the Algorithm Selection Library (ASlib[3]). However, since we want to select parallel portfolios for commonly used parallel architectures, that is, at least a quadcore processor, we focus on scenarios with at least four algorithms, that is, excluding the *CSP-2010* scenario with only two algorithms. Furthermore, 7 of the 12 ASlib scenarios are built upon SAT solvers. We focus only on the newer and larger SAT scenarios from 2012, that is, excluding three *SAT11-**scenarios. Thus, we conduct an extensive experimental study to assess the efficacy of *PASU* on eight scenarios,

[3] `www.aslib.net`

namely, *ASP-POTASSCO*, *MAXSAT12-PMS*, *PREMARSHALLING-ASTAR-2013*, *QBF-2011*, *SAT12-ALL*, *SAT12-INDU*, *SAT12-HAND* and *SAT12-RAND*.

As specified in the scenarios, we evaluated our approaches using a 10-fold cross validation. Since *PASU* cannot solve instances that are not solved by any selectable algorithm, we removed such instances from the test sets; see Table 5.7 in Section 5.5.

Score Prediction Model. Since our approach assumes independent performance estimations for each algorithm, we cannot use approaches as those underlying *SATzilla'11-like* (see Section 5.4.5). However, first results on ASlib show that a performance estimation approach with Random Forest Regression performs well on all these scenarios. Also, Hutter, Xu, Hoos, and Leyton-Brown (2014c) show that Random Forest Regression is well suited for predicting algorithm runtimes. Such an approach is also supported by *claspfolio* 2 and we will use it for the selection models $\mathcal{M}$. The Random Forest Regression is implemented via the Python Package *sklearn* (Pedregosa et al., 2011). This implementation has three further parameters; we set them to the default values used in *sklearn*: (i) the number of trained regression trees is 10, (ii) the maximal number of features at each split is the square root of the number of features and (iii) the minimal number of samples in each leaf is 2.

Approaches and Parameters. We compare the two presented approaches, that is, the *baseline* approach with the selection of the k algorithms with the best predicted performances and the *PASU* approach with consideration of the score distribution. For *PASU*, we consider five commonly used percentiles $q \in \{0, 25, 50, 75, 100\}$, that is, the minimum, the lower quartile, median, the upper quartile and the maximum. $q = 0$ can be interpreted as a portfolio with optimistic performance estimation, that is, the best predicted performance. In contrast, $q = 100$ refers to a pessimistic performance estimation, that is the worst predicted performance. $q \in \{25, 50, 75\}$ are gradations between these two extremes.

For the training of our models, we used $n = 42$ models $\mathcal{M}$, a feature subset size of $v = 70\%$ of the original feature size, and bootstrapped training data per model of $k = 70\%$ of the original training data size. In the construction method for the parallel portfolio, we sampled $r = 1\,000$ performance scores to estimate the score distribution of each parallel portfolio.

Overhead. Normally, hardware bottlenecks induce some overhead for running parallel portfolios. However, since this overhead is hardware-dependent and we have no access to the hardware used to generate the ASlib scenarios, we will not consider such overhead in our experiments. Therefore, the performance presented in our experiments can be seen as a theoretical lower bound (under the assumption that the performance is minimized).

Statistical Testing. With growing size of parallel portfolios, the performance will improve independently of the underlying approach because the portfolio is constructed greedily. However, the performance of a parallel portfolio is bound by the performance of the maximal portfolio, that is, a portfolio with all selectable algorithms.[4] Since we use algorithm selection, the per-

[4] We do consider instance feature computation costs in the performance of the maximal portfolio. Therefore, the performance of the maximal portfolio is worse than a perfect oracle selector that does not consider the feature costs.

formance of the maximal portfolio should be reached with much smaller portfolios, and we can save parallel resources with smaller portfolios. In the best case, the portfolio would have a size of 1, that is, the best per-instance algorithm was always selected. Therefore, we prefer approaches with the smallest parallel portfolios whose performance is indistinguishable from that of the maximal portfolio. We use a statistical test, that is, the Mann-Whitney U test with significance level 0.05, to verify that the performance of a given per-instance portfolio is not worse than the performance of the maximal portfolio.[5]

6.3.2 Results

Figures 6.2 and 6.3 show the PAR10 performance trend over the size of the portfolio. A vertical line marks the smallest portfolio with a indistinguishable performance to the maximal portfolio (according to the Mann-Whitney U test). Hence, an approach is better if a vertical line is further left. To prevent overlapping of the vertical lines, they are minimally shifted to the right if several approaches have the same value.

In *MAXSAT12-PMS*, *PASU(*$q = 25$*)* and in *SAT12-HAND*, *PASU(*$q = 50$*)* selects parallel portfolios with smaller size than the *baseline* approach. However, *PASU(*$q = 0$*)* needs larger portfolios in all scenarios except *PREMARSHALLING-ASTAR-2013* and *MAXSAT12-PMS*, and *PASU(*$q = 100$*)* in all except *PREMARSHALLING-ASTAR-2013* and *SAT12-ALL*. In all scenarios, *PASU(*$q = 50$*)* is at least as good as the *baseline*.

6.3.3 Discussion

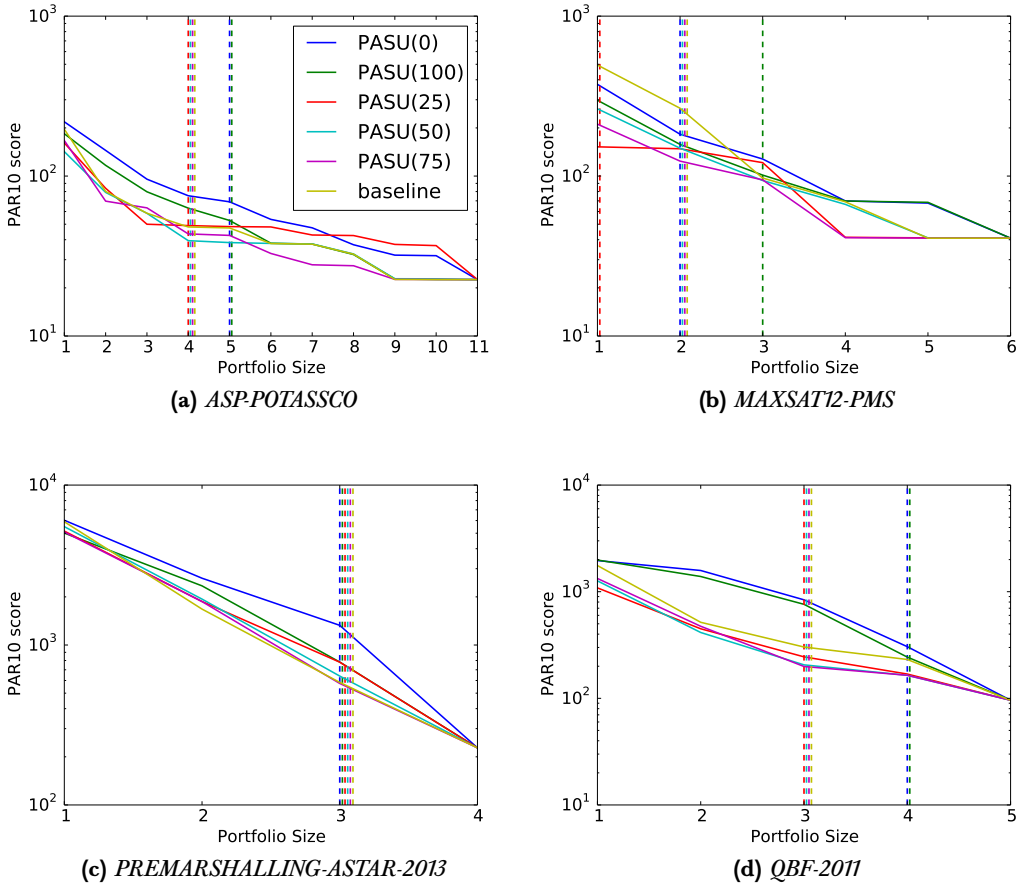
First, we note that *PASU(q)* improved the performance of the *sequential claspfolio* 2 in comparison to *baseline* on all scenarios for $q = 50$, and in 7 out of 8 scenarios with $q = 25$. This is surprising, because the used Random Forest Regression for the performance estimation models already uses an ensemble of regression trees and *PASU* adds only a further ensemble level on top of this. However, Random Forest Regression models average the predictions of each regression tree. In contrast, *PASU* does not average performance scores, but optimizes a given percentile of the score distribution.

In the sequential case and in the parallel case, totally pessimistic ($q = 0$) and optimistic ($q = 100$) performance predictions induce a worse performance than more intermediate predictions. The best choice between *PASU(*$q \in \{25, 50, 75\}$*)* and the *baseline* approach depends on the algorithm selection scenario.

We note that on *SAT12-HAND*, *PASU* and also the *baseline* approach found portfolios with sizes of $20-23$ that are indistinguishable from the maximal portfolio. However, these portfolios have a PAR10 score that is approximately five times greater than the PAR10 score of the maximal portfolio. Since the Mann-Whitney U test is based on ranks of performances, the average performance difference can be substantial, because of outliers.

We observed in preliminary experiments that apart from q, also the parameters for the feature subset size and for the bootstrapped training data size influence the performance of *PASU*. The

[5] We cannot apply a permutation test to differentiate the performance of greedily constructed portfolios. If we incrementally construct our parallel portfolio, the performance of the portfolio will improve after each iteration. Hence, the performance of a subset of a portfolio $P' \subset P$ is always dominated by P, that is, if $P' \subset P$ then $m(i,P') \geq m(i,P)$ holds for every instance $i \in I$. A permutation test will always return that P' is significantly worse than P.



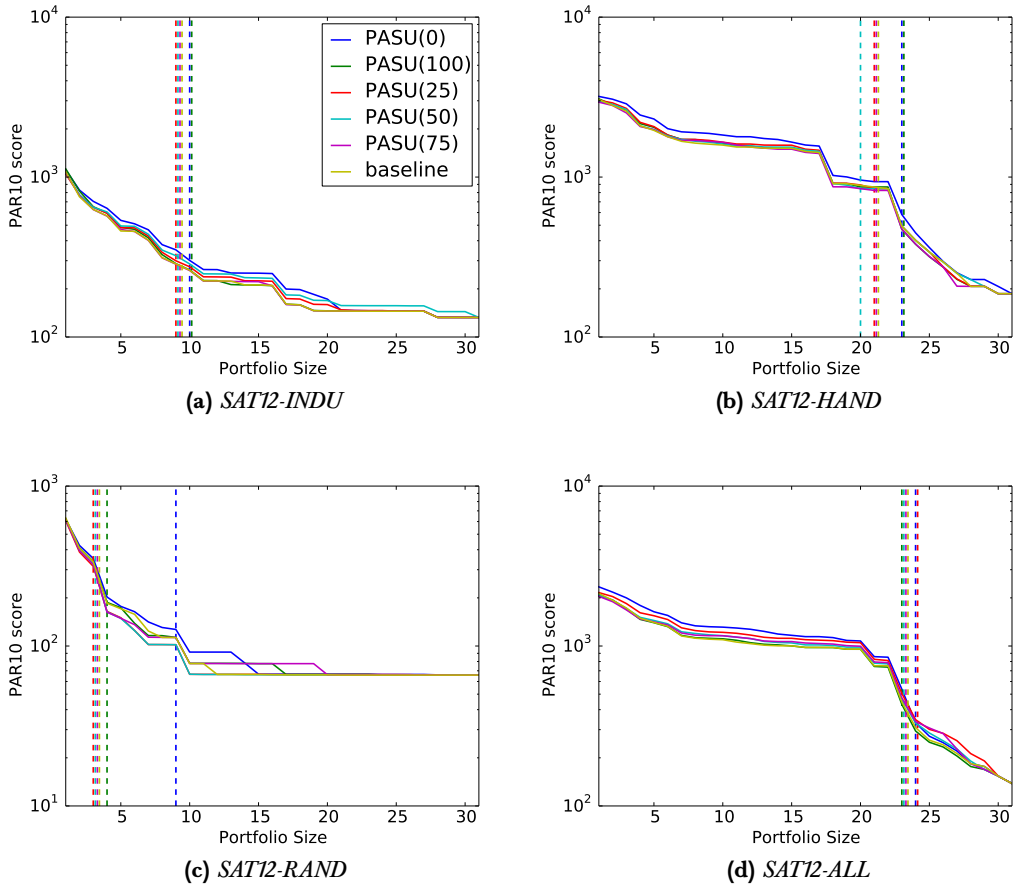
(a) *ASP-POTASSCO* **(b)** *MAXSAT12-PMS*

(c) *PREMARSHALLING-ASTAR-2013* **(d)** *QBF-2011*

Figure 6.2: Diverse Scenarios - PAR10 Performance (without unsolvable instances) over size of portfolio. Vertical lines indicate that there is no statistical difference between the performance and the optimal performance of the maximal portfolio (according to a Mann-Whitney-U-Test with significance level 0.05).

influence differed between the individual algorithm selection scenario. For these experiments, we decided to use commonly used values for these parameters (see above). A further degree of freedom is the approach used for predicting the performance scores – in our case: Random Forest Regression. We therefore expect that the performance of *PASU* can be further increased by using automatic algorithm configuration methods, such as *SMAC* (Hutter et al., 2011a), to configure q and the other parameters.

We also investigated in preliminary experiments whether using a higher number of performance estimation models (n) would influence the performance of *PASU*. The number of used models cannot be increased without limit because *PASU* obtains a performance prediction from each trained model when selecting an algorithm or parallel portfolio. Even though, performance predictions are normally cheap, using too many predictions increase the overhead for the selection and hence reduce the time for the actual solving process. In our experimental setup, we trained a Random Forest Regression model for each bootstrapped sampled training



(a) *SAT12-INDU* **(b)** *SAT12-HAND*

(c) *SAT12-RAND* **(d)** *SAT12-ALL*

Figure 6.3: SAT Scenarios - PAR10 Performance (without unsolvable instances) over size of portfolio. A vertical line marks the first portfolio with a performance indistinguishable to the maximal portfolio (according to a Mann-Whitney-U-Test with significance level 0.05).

set and each algorithm. The performance of *PASU* began to drop between 10 and 20 models (depending on the scenario) since the uncertainty estimation was too rough. On the other side, we have not observed substantial changes of the performance between 20 and 100 models.[6]

6.4 Empirical Performance Comparison against *aspeed*

As already shown in Section 5.5, *aspeed*'s static algorithm schedules perform sometimes better than the per-instance selection approaches in *claspfolio* 2 in the sequential case. Now, we further investigate the difference between the per-instance selected parallel portfolio of *claspfolio* 2 with *PASU* and the static parallel algorithm schedules computed by *aspeed*.

[6] 100 model evaluations were still negligible in comparison to the runtime cutoff and have not produced overheads that induced timeouts when solving problem instances.

6.4.1 Setup.

Since *PASU* with $q = 50$ performed best on average in our experiments reported in Section 6.3, we fixed the parameter accordingly. For *aspeed*, we limited the memory usage to 4 GB (never encountered) and the runtime to 600 CPU seconds (encountered in all SAT scenarios because of the large number of solvers in these scenarios[7]). *aspeed* relies on the ASP grounder *gringo* 3.0 and the ASP solver *clasp* 2.2. Furthermore, we assessed the performance of the *single best* in parallel setting, that is, a static selection of the n best solvers on the training data. The *single best* and *aspeed* do not rely on instance features so that the costs to compute instance features is only considered for *PASU* and *baseline*.

The overhead due to hardware bottlenecks for parallel portfolio runs is again not taken into account. Once again, we used the 10-cross validation given by the scenarios and report PAR10 performance that does not include unsolvable instances.

6.4.2 Results.

Table 6.1 shows the PAR10 performances of *PASU*($q = 50$), *single best*, the *baseline* approach and *aspeed* with parallel portfolio sizes of 1, 2, 4 and 8. As expected, the performance of all three approaches improves with the size of the portfolio, except for *aspeed* at *MAXSAT12-PMS* with a portfolio of size 2. Since parallel portfolios of *aspeed* are not incrementally constructed as done in *PASU* and *baseline*, *aspeed* can select other solvers for each portfolio. Therefore, it is in theory possible that a larger portfolio can generalize worse on a test set than a smaller portfolio. However, this is the only time we observed this.

We note that *single best* and *aspeed* perform in particular well on scenarios with large costs for feature computation. For example, *aspeed* is the best approach on *SAT12-INDU* with a portfolio size of 8 and *single best* is also better than *baseline* and *PASU*. Furthermore, *single best* is the best approach on *SAT12-RAND*. As already observed in Section 5.5, *claspfolio* 2 would need further tuning of its parameter to perform better than the *single best*. On *PREMARSHALLING-ASTAR-2013*, *single best*, *baseline* and *PASU* have the optimal performance with a portfolio size of 4 since the scenario contains of 4 selectable algorithms and the cost of the feature computation is negligible.

Independent of the portfolio size, *PASU*($q = 50$) is the best approach on *ASP-POTASSCO* and *QBF-2011*. *aspeed* performs consistently the best on *SAT12-HAND*. On the other four scenarios, the best approach depends on the size of the portfolio. On *MAXSAT12-PMS*, *SAT12-ALL* and *SAT12-INDU*, *PASU*($q = 50$) has better performance on smaller portfolios and *aspeed* gets better for larger number of processing units. The converse situation is observed for *PREMARSHALLING-ASTAR-2013*.

6.4.3 Discussion.

The unique characteristic of the SAT scenarios in comparison to the other scenarios is that there are many more algorithms available. In all other scenarios, a portfolio with a size of 4 or 8 nearly uses the entire range of possible algorithms. Since the parallel schedule produced by *aspeed* can run several algorithms on one processing unit, *aspeed* may take better advantage

[7] In all cases of timeouts of *aspeed*, *aspeed* nevertheless returned a list of possible schedules with their optimization scores and we took the best of these schedules.

Portfolio Size:	1	2	4	8
ASP-POTASSCO				
single best	534	177	72	60
baseline	196	80	48	**32**
aspeed	367	204	84	50
PASU($q = 50$)	142	79	39	**32**
MAXSAT12-PMS				
single best	2111	1635	1197	—
baseline	494	265	69	—
aspeed	280	365	**44**	—
PASU($q = 50$)	263	149	66	—
PREMARSHALLING-ASTAR-2013				
single best	7002	4903	**227**	—
baseline	5896	1677	**227**	—
aspeed	1969	588	484	—
PASU($q = 50$)	5495	1936	**227**	—
QBF-2011				
single best	9172	3344	674	—
baseline	1759	516	231	—
aspeed	1507	927	310	—
PASU($q = 50$)	1263	414	**164**	—

Portfolio Size:	1	2	4	8
SAT12-ALL				
single best	2967	2872	2165	1727
baseline	2128	1932	1492	1119
aspeed	2672	2002	1277	**668**
PASU($q = 50$)	2083	1927	1510	1197
SAT12-INDU				
single best	1360	879	547	285
baseline	1083	752	572	313
aspeed	1793	1042	544	**259**
PASU($q = 50$)	1067	773	605	349
SAT12-HAND				
single best	3929	3885	2775	2093
baseline	3050	2805	2091	1629
aspeed	2296	1757	1166	**505**
PASU($q = 50$)	2980	2845	2090	1676
SAT12-RAND				
single best	568	383	143	**82**
baseline	631	408	185	113
aspeed	681	452	180	141
PASU($q = 50$)	620	369	163	102

Table 6.1: Comparison of *PASU*, a static *single best* selection, the *baseline* approach with a Random Forest Regression and *aspeed*'s static algorithm schedules on PAR10 scores without unsolvable instances. The best performance per scenario is bold. If the number of selectable algorithms is smaller than the parallel portfolio size, we marked the corresponding entry with "—".

of the additional processing units than the approach underlying *PASU*. Only on *SAT12-RAND* from the SAT scenarios, *PASU* and also the *baseline* approach perform better than *aspeed* on eight processing units. We suspect that *aspeed* has a harder task here, because of the stochastic local search SAT solvers which are dominant in the state-of-the art SAT solving for randomly generated instances. We already made the same observation on *SAT11-RAND*, see Section 4.4.

A future step is to combine the per-instance selection of parallel portfolio based on *PASU* with the algorithm schedules of *aspeed*, since both were found to have weaknesses and strengths on different scenarios. The straightforward approach consists of a combination of pre-solving schedules found by *aspeed* before *PASU* selects the portfolio.

6.5 Conclusion

Overall, our new approach to select instance-specific parallel portfolios, *PASU*, shows promising results on a diverse set of different algorithm selection scenarios from the Algorithm Selection Library (ASlib). On a commonly used quad-core machine, *PASU* reduced the PAR10 in comparison to the sequential counterpart by a factor of $1.4 - 24.20$ (on average by 5.95). In some

scenarios, *PASU* performed better than the *baseline* approach to select parallel portfolios, that is, the selection of the k algorithms with best predicted performances. This is due to the fact that *PASU* was constructed in a way to overcome certain drawbacks of the *baseline* approach which do not apply in all scenarios. One advantage of *aspeed* over *PASU* is that the runtime of *aspeed*'s algorithm runs is more strongly limited so that runs can be prematurely aborted if the probability is low that they will the instance with more time. Therefore, *aspeed* performed better than *PASU* in particular on scenarios with larger sets of selectable algorithms, such as the SAT12 scenarios, since *aspeed* can try several algorithms for a short amount of runtime.

Apart from a combination of *aspeed* and *PASU*, we believe that the performance of *PASU* can be further improved with the help of automatic algorithm configuration. *PASU* was designed and implemented in *claspfolio* 2 on level 4 of *programming by optimization* (Hoos, 2012), that is, *PASU* has adjustable parameters for all design choices that could not be justified prematurely. This was the right choice since the best configuration of the parameter q of *PASU* differed between different scenarios and should be adjusted depending on the scenario. Nonetheless with a default value of $q = 50$, *PASU* already showed convincing results on all considered scenarios.

PASU relies only on one requirement for the underlying algorithm selector: the selector has to return an independently computed score for each algorithm in the base portfolio. Therefore, *PASU* is not limited to *claspfolio* 2 but can also be applied to other algorithm selectors, for example *SATzilla*'09 (Xu et al., 2009), to enable them to select also parallel portfolios.

7 Empirical Performance Comparison

So far, we compared all presented approaches on different benchmark sets available at the time our respective studies were carried out. In this chapter, we apply ACPP's portfolio, *aspeed* and *claspfolio* 2 to two Answer Set Programming (ASP) benchmarks with orthogonal properties to outline their respective strengths and weaknesses. We follow the *ASP Practitioner's Guide* from Silverthorn et al. (2012) by investigating in which use cases which approach should be applied. However, in contrast to Silverthorn et al., we do not focus on sets of similar problems but on problems with two orthogonal use cases. Furthermore, Silverthorn et al. considered only sequential solvers, whereas we also consider parallel solvers.

First, we use the benchmark suite of Alex Rudolph's board game *RICOCHET ROBOTS*[1] on which Gebser et al. (2013) showed different ways to model and solve the problem using ASP. We focus on the decision problem variant using the advanced encoding. Since all problem instances are generated using one encoding, this benchmarks represents the use case of solving a particular homogeneous problem with ASP. Second, we use the algorithm configuration scenario *ASP-POTASSCO* from the Algorithm Selection Library (ASlib[2]) which consists of a heterogeneous set including several applications and instances from ASP Competitions. This second benchmark represents a use case, where an ASP solver is applied to different problems and needs a robust performance to perform well on all kinds of instances.

7.1 Experimental Setup

In the following, we present the experimental setup we used to empirically compare the performance of our systems.

7.1.1 Instance Sets

RICOCHET ROBOTS. We generated 400 instances on the original board and randomly varied the start positions of the four robots, the color of the goal, the position of the goal on the board and the maximal horizon (between 10 and 30 steps) of the plan that has to be found. We used the *advanced* encoding by Gebser et al. (2013) and grounded all instances with *gringo* 3.0 (Gebser et al., 2011).

ASP-POTASSCO. The *ASP-POTASSCO* benchmark set consists of 2589 instances from 105 problem classes and includes instances from the ASP competitions organized in 2007 (SLparse track), 2009 (with the encodings of the Potassco group) and 2011 (decision NP-problems from the system track), as well as several instances from the ASP benchmark collection platform

[1] http://en.wikipedia.org/wiki/Ricochet_Robot
[2] www.aslib.net

asparagus[3]. The test set of the *ASP-POTASSCO* (see Subsection 7.1.2) set is also part of the ASlib.

7.1.2 Training Test Split

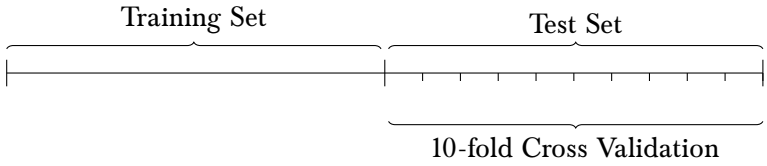


Figure 7.1: Training and test split of the instances in the scenarios.

We split the instance sets in the same way as in Section 5.4, illustrated in Figure 7.1. Each instance set was randomly split in two equal parts, the training set and the test set. The test set was further randomly split in ten equal parts to be used in a 10-fold cross validation. All configuration experiments were performed on the training set. On the test set, our systems were unbiasedly assessed using cross validation.

7.1.3 Systems

All of our presented approaches (ACPP, *aspeed* and *claspfolio* 2) are based on a portfolio of algorithms, that is, a set of different solvers or solver configurations. To compare them in a fair way, all approaches use the same portfolio. The portfolio was constructed using the ACPP approach *parHydra* (see Subsection 3.2.1.3). However, the approach was slightly modified in the following way to generate arbitrary many constituent algorithms: (i) the method stopped when the portfolio improved by less than 1% on the PAR10 score in comparison to the previous iteration and (ii) the constituent algorithms ran sequentially using the same adaption of the performance metric after each iterations as proposed by *hydra* (Xu et al., 2010) and not in parallel as in Chapter 3. In this way, the method resembles more the original *hydra* method. The configuration process performed by *SMAC* (2.06.01; Hutter et al. (2011a)) in each *hydra* iteration was allocated 120 000 CPU seconds, that is, 200 times the target algorithm cutoff time, and 10 independent repetitions, from which the result with the best PAR10 score on the given training set was selected.

Since the portfolio might suffer from over-tuning on the training set, we could not train our portfolio systems on the training set, because the performance estimation would be too optimistic. Therefore, our portfolio systems were unbiasedly assessed on the test set using a 10-fold cross validation.

In our experiment, we study the following sequential systems and parallel systems with four processing units, because in most mainstream systems, quad-core processes are still dominant. All systems are based on configurations of the state-of-the-art ASP solver clasp (2.1.3):

- *Default-SP*: the default sequential configuration of *clasp*;

[3] `http://asparagus.cs.uni-potsdam.de`

- *single best*: the best known sequential single configuration of *clasp* on the training set; the set of configurations we considered includes the default configuration, an expert[4] configuration, and all configurations that are part of the configured portfolio;
- *parHydra*: a static parallel portfolio using the first four constituent components of the previously described portfolio (although the configuration may produced a larger portfolio); in this way modification (i) of *parHydra* is revoked; see Chapter 3;
- *aspeed* in its sequential and parallel version; see Chapter 4;
- *claspfolio* 2: in its sequential version with the default configuration based on the *SATzilla'11-like* approach including pre-solving schedules by *aspeed*;
- *claspfolio* 2+*PASU*($q = 50$) in its sequential and parallel version where the parameter for the score distribution percentile q is set to 50; for selection, *claspfolio* 2 relies on Random Forest Regression as described in Section 6.3; for instance features, *claspre* generated 38 static features and 25 dynamic features after each of four restarts (*claspre(s+d)*); for further details see Chapters 5 and 6.

7.1.4 Hardware and Software

All our experiments were performed on the Zuse computer cluster in Potsdam with dual Intel Xeon E5520 quad-core processors (2.26 GHz, 8192 KB cache) and 48 GB RAM per node, running Scientific Linux (2.6.18-308.4.1.el5). Each algorithm run was limited to a runtime cutoff of 600 CPU seconds and to a memory cutoff of 6 GB.

7.2 Results

The portfolio found by *SMAC* for *RICOCHET ROBOTS* consists of 15 constituent configurations and for *ASP-POTASSCO* of 11 constituent configurations, see Appendix C. On the one hand, we expected that the portfolio for *RICOCHET ROBOTS* would be smaller than the portfolio for *ASP-POTASSCO*, because *ASP-POTASSCO* consists of several problems (more than 100 problem classes) and *RICOCHET ROBOTS* only of one. Under the assumption that the best configuration differs for each problem class, a portfolio for *ASP-POTASSCO* could have more than 100 components. On the other hand, as already observed by Schneider and Hoos (2012), configuration on heterogeneous instance set is more challenging than on homogeneous sets. Since *ASP-POTASSCO* is a very heterogeneous set, *SMAC* was not able to find further improving configurations within the given time budget.

Table 7.1 shows the PAR10 performance, the number of timeouts and the PAR1 performance of our systems on the *RICOCHET ROBOTS* test set. Here, *single best* is the configuration found in the sixth iteration of the portfolio construction. It solved all instances and improved the performance by a factor of 55.5 on PAR10. The sequential *aspeed* (using only 2 of the 15 available *clasp* configurations) and sequential *claspfolio* 2+*PASU* both made an incorrect decision for one instance leading to a timeout. The default *claspfolio* 2 even had three timeouts (running at least once 10 of the 15 available *clasp* configurations. The parallel portfolio of

[4]Reference Benjamin Kaufmann, main developer of *clasp*

	PAR10	#TOs	PAR1
Sequential			
Default-SP	961.97	30	151.0
single best	17.32	0	17.32
aspeed	46.99	1	16.99
claspfolio 2+*PASU*($q = 50$)	50.59	1	23.59
claspfolio 2	110.44	3	29.44
Parallel with 4 **Processing Units**			
parHydra	73.96	2	19.96
aspeed	16.90	0	16.90
claspfolio 2+*PASU*($q = 50$)	13.21	0	13.21
oracle	7.04	0	7.04

Table 7.1: Cross validated performance on *RICOCHET ROBOTS*'s test set regarding wall-clock time in seconds.

parHydra only includes the first four components of the portfolio; hence, it does not include the *single best* configuration. Therefore, *parHydra*'s performance was even worse than the sequential systems, except *Default-SP*. The parallel *aspeed* and *claspfolio* 2+*PASU* both solved all instances and slightly improved the PAR1 performance.

	PAR10	#TOs	PAR1
Sequential			
Default-SP	1374.18	287	176.50
single best	880.55	183	116.87
aspeed	774.72	149	152.93
claspfolio 2+*PASU*($q = 50$)	497.15	101	75.66
claspfolio 2	483.50	98	74.54
Parallel with 4 **Processing Units**			
parHydra	552.01	114	76.31
aspeed	458.89	93	70.79
claspfolio 2+*PASU*($q = 50$)	417.16	85	62.4
oracle	400.17	82	57.98

Table 7.2: Cross validated performance on *ASP-POTASSCO*'s test set regarding wall-clock time in seconds.

Table 7.2 shows the same performance metrics on the test set of *ASP-POTASSCO*. Again, the *single best* configuration of *clasp* outperformed the default configuration by solving 104 more instances. This time, the *single best* turned out to be the expert configuration constructed by Benjamin Kaufmann and used by *clasp* in the 2013 ASP Competition. In the sequential case, *aspeed* solved 34 more instances (running at least once 10 of the 11 available *clasp* configurations) and *claspfolio* 2+*PASU* 48 additional ones. Using *claspfolio* 2 in its default configuration with *aspeed*'s pre-solving schedule (running at least once all 11 available *clasp* configurations) resulted in solving additional 3 instances. In the parallel case, the ACPP portfolio by *parHydra*

had a substantially better performance than its configured sequential counterpart, *single best*, but *parHydra* had the worst performance of our parallel systems. Also the performance of *parHydra* was worse than sequential *claspfolio* 2. As in the case of the parallel schedules, *aspeed* is able to solve more instances than *parHydra*'s portfolio and all sequential solvers. The parallel version of *claspfolio* 2+*PASU* is able to solve the most instances and has only 3 timeouts more than the perfect algorithm selector, *oracle*.

7.3 Discussion

The results indicate that portfolio-based systems perform well in comparison to non-portfolio systems on heterogeneous instance sets, such as *ASP-POTASSCO*. Nevertheless, the portfolio-based systems have nearly no or only a small advantage in comparison to an automatically configured version of plain *clasp* in case of a homogeneous instance set. Silverthorn et al. (2012) came to a similar conclusion, based on their experiments on three other homogeneous instance sets. This can have several reasons:

- There exists a single configuration of *clasp* that performs well on all instances of a homogeneous set so that even a portfolio is mainly dominated by this one configuration; results in literature regarding algorithm selection (see, for example, Xu et al. (2010)) indicate that such configurations do not exist for heterogeneous sets.
- Instance features are very similar across a homogeneous set of instances so that an algorithm selection approach has a harder task to reliably differentiate the instances based on their features; specialized instance features for a certain application could improve the performance of algorithm selection in this case.

In practice, one question remains unanswered: how to identify homogeneous and heterogeneous instance sets. As a rough guideline: an instance set is homogeneous if all instances belong to *one* problem class (or application); and an instance set is heterogeneous if it consists of instances belonging to several problem classes. However, this rough guideline does not hold always. For example, if an instance set consists of small and large instances of the same problem, normally there is a well performing configuration for the small instances (for example, a more conservative deletion strategy of learned clauses) and another configuration for the large instances (for example, a more aggressive deletion strategy).

First approaches to assess the homogeneity of instance sets in the context of algorithm configuration were proposed by Hutter et al. in a qualitative way (Hutter, Hoos, & Leyton-Brown, 2011b) and by Hoos and Schneider in a quantitative way (Schneider & Hoos, 2012). One of their conclusion was that "*more homogeneous instance sets are more amenable to automated algorithm configuration*" (Schneider & Hoos, 2012). Furthermore, one of their measures "*helps to assess the specific potential of portfolio-based approaches in a given configuration scenario*". Unfortunately, both approaches are based on computational expensive runtime collections of hundreds or thousands of different random configurations on an instance set. Therefore, their practical applicability is limited. For practical usage and in future work, in case benchmarking all proposed systems is practically not feasible, a tool is needed that assesses effectively the homogeneity of an instance set and characteristics of the feature space to conclude which of the presented systems should be used.

8 Conclusion and Discussion

In this work, we tackled a widespread problem: Users often have no idea how to choose a well-performing strategy for solving their applications. This includes the choice between several solvers and the configuration of the chosen solver. On the one hand, beginners have no idea, because they literally know nothing about the strengths and weaknesses of different solving strategies; on the other hand, even experts often have knowledge and deeper understanding only either of solving strategies or applications. However, experts for both, solving strategies and application, are seldom available so that users rely on default configurations of a randomly picked solver in the worst case. As previously reported in literature (Hutter et al., 2009; Ansótegui et al., 2009; López-Ibáñez et al., 2011; Hutter et al., 2011a) and confirmed as a by-product in this work, the default configuration of algorithms can be a lot worse than specialized solving strategies (for example, a configured solver); sometimes by more than one order of magnitude.

So, the question arises how to improve the robustness of algorithms to enable users to solve their problems effectively with out-of-the-box solvers. In this work, we tackled this problems by relying on meta solving techniques, such as algorithm configuration, algorithm schedules and algorithm selection. Under consideration of the increasing importance of parallel computation, we also extended all our approaches to parallel solving and investigated its benefits.

We demonstrated that:

1. sequential algorithms can be combined automatically and effectively into parallel portfolios by using algorithm configuration, – we call this approach Automatic Construction of Parallel Portfolios (ACPP);

2. ASP formulations and a powerful ASP solver (*clasp*) can be used to compute sequential and parallel algorithm schedules – this is the basis for our *aspeed* procedure;

3. an effective and modular algorithm selection solver can be build upon automatic portfolio construction methods, cheap-to-compute instance features, algorithm-schedule-based pre-solving techniques and algorithm selection approaches – we demonstrate this with our *claspfolio* 2framework;

4. algorithm configurators can be effectively extended to select a parallel portfolio of algorithms – we demonstrate this with our Parallel Algorithm Selection with Uncertainty (*PASU*) approach.

These approaches have several advantages in comparison to using an arbitrary algorithm out-of-the-box:

- basically eliminated the need for human experts to choose an appropriate solving strategy since our approaches do this automatically;

- improved robustness, so that an algorithm can be efficiently used on large and diverse sets of instances;
- automatic methods for developers to construct robust sequential and parallel algorithms.

All our approaches are not specific to a certain problem class, such as SAT or ASP, but can be applied to arbitrary problems. In particular, we expect that our approaches are effective for *NP*-hard problems, because poorly chosen solving strategies can have exponential longer runtimes than the well chosen strategies. However, the application to polynomial problems is also possible, since polynomial algorithms are often parameterized and they can be adjusted to given problem characteristics (see, for example, basic linear algebra procedures (Whaley, Petitet, & Dongarra, 2001), database systems (Diao, Eskesen, Froehlich, Hellerstein, Spainhower, & Surendra, 2003), sorting (Li, Garzarán, & Padua, 2007) or compilers (Cavazos & O'Boyle, 2005)).

8.1 When to apply which method?

In the end, we have to discuss when to apply which of our presented methods. Our methods have in common that we assume that there exist algorithms implementing *complementary* strategies (that is, no strategy dominates all other strategies) from which we can choose. If the available algorithms are not complementary, there is a dominant algorithm outperforming all other algorithms on all kind of instances and we can use this dominant algorithm after applying algorithm configuration to identify it. In this case, there is no additional benefit in using portfolios of algorithms (that is, a set of algorithms) as long as they do not interact among themselves by sharing of intermediate results (for example, clause sharing in SAT or sharing of optimization bounds in MAXSAT). However, empirical results, also in this work, indicate that there does not exist such a dominant algorithm over all kinds of *NP*-hard problems.

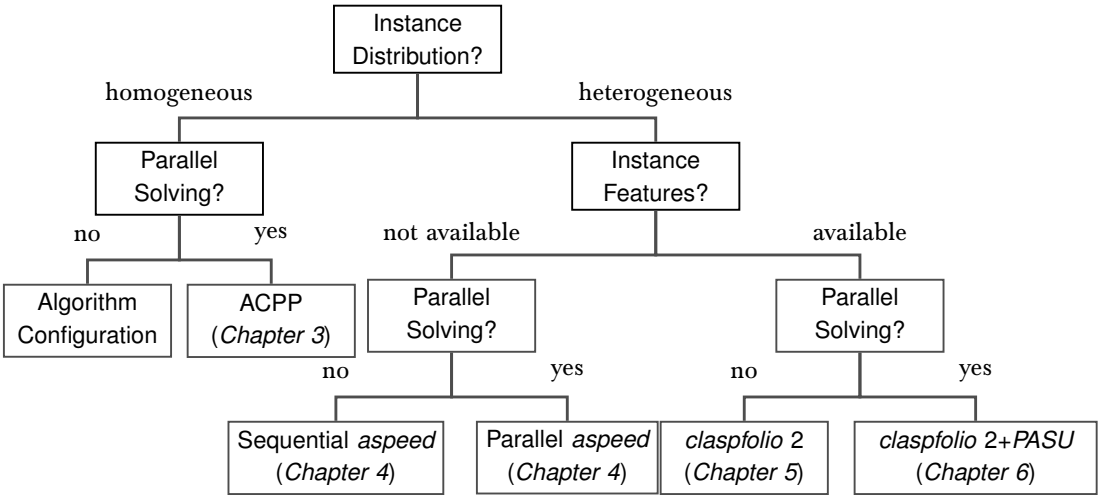


Figure 8.1: High-level guideline for selecting meta-algorithmic approach to be used in a given situation.

These complementary strategies can be available in the form of parameterized algorithms or portfolios of different algorithms. Considering the availability of complementary solving strategies, the question still remains which of our methods should be used. Figure 8.1 visualizes a simplified guideline that hold in most use cases. The first question considers whether the set of instances to be solved is homogeneous or heterogeneous, as discussed in Chapter 7.

If a user wants to solve similar instances from a certain application, for example, travelling salesman with similar properties such as number of cities, the instances are most likely homogeneous. Then, algorithm configuration is the right choice to find a well-performing specialized solving strategy for the application. If parallel resources are available, our ACPP approach can be used to automatically construct a parallel solver based on sequential solver.

If the user plans to solve instances belonging to many problem classes, such as different action planning task in robotic scenarios, the instances are most likely heterogeneous, and a portfolio-based solver (such as algorithm scheduler, algorithm selector or a parallel porfolio solver) will probably perform better than a configured non-portfolio solver. The choice between static algorithm scheduling and per-instance algorithm selection (supported by algorithm schedules) mainly depends on the availability of informative instance features. If instance features are not available, *aspeed* can be used to find effective sequential or parallel algorithm schedules. If cheap-to-compute instance features with information content are available, algorithm selection systems supported by pre-solving schedules, such as *claspfolio* 2, should be preferred over static schedules in many cases, since it selects a presumable well-performing instances per-instance.

As we have seen in the empirical performance analysis in this work (Chapter 4), there are exceptions to this guideline. For example, there exist algorithm selection scenarios where *aspeed* performs better than algorithm selectors, such as *claspfolio* 2. Furthermore, as we have done in Chapter 7, *claspfolio* 2 can also be applied in the homogeneous use case and probably, it will constantly select the best solver in the portfolio.

8.2 Future Work

Although, according to our empirical results, the methods we have presented in this thesis improve state-of-the-art solving, there is plenty of room for further work.

So far, we have considered only one type of performance metric, that is, runtime. Another typical performance metric is solution quality, that is, given a fixed time budget, an algorithm has to find a solution with the highest quality. This quality can be, for example, the cost of a round trip through a set of cities (traveling salesman problem). Algorithm configuration and algorithm selection are defined on an arbitrary performance metric (such as number of timeouts, average runtime or penalized average runtime) and have already proved to be also effective for average quality as performance metric (see, for example, Hutter et al. (2009) for algorithm configuration or (Bischl, Mersmann, Trautmann, & Preuß, 2012; Amadini, Gabbrielli, & Mauro, 2014) for algorithm selection). However, all these approaches have not yet considered effective pre-solving schedules. When an optimization algorithm is an anytime algorithms, that is, on any point in time, the algorithm can be interrupted and will return the best solution so far, this further degree of freedom complicates the problem of optimising schedules. It is not anymore a binary decision whether an algorithm solved an instance or not as we studied in Chapter 4.

Another direction of future work consists of considering interactions between algorithms. For example, it is well-known that the performance of parallel CDCD-based SAT solvers can be significantly improved by using clause sharing (see, for example, Hamadi et al. (2009a)). In Chapter 3, we have considered clause sharing as an additional step of ACPP, but we have not yet tightly integrated it in ACPP by considering the effect of clause sharing while constructing the portfolio. For sequential algorithm schedules, Malitsky et al. (2013b) export and import learned clauses between two SAT solvers in succession. However, they have not considered the influence of the alignment of the algorithms in the schedule when using this kind of clause sharing. For algorithm selection, Malitsky et al. (2012) also considered parallel solvers with clause sharing in their selected parallel portfolios. However, the solvers do not interact with each other. At least in parallel SAT solving, clause sharing introduces non-deterministic runtime behavior[1], which makes it harder to collect reliable training data for our methods.

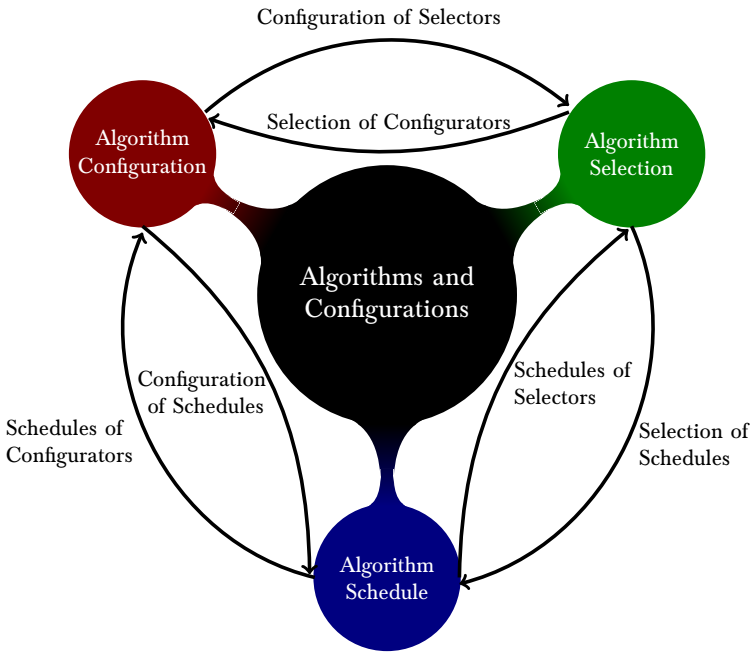


Figure 8.2: Overview of meta-algorithmic techniques and possible combinations.

Another direction for future work is the combination of algorithm configuration, scheduling and selection. So far, we combine them only in two aspects: (i) we use algorithm configuration to construct base portfolios for algorithm schedules or algorithm selection; (ii) algorithm schedules are used for pre-solving before algorithm selection is used. However, there exist further possible combinations, see Figure 8.2. Kadioglu et al. (2011) combined algorithm schedules and algorithm selection in a way that per-instance schedules are selected. Another combination is the configuration of schedules by Seipp, Sievers, and Hutter (2015). For example, algorithm configuration can be seen itself as an (black-box) optimization process and different algorithm

[1]Deterministic parallel SAT solving with clause sharing (see, for example, Hamadi, Jabbour, Piette, and Sais (2011)) have not yet demonstrated to yield state-of-the-art performance.

configurators have shown to have strengths and weaknesses on different scenarios (see, for example, results of the Configurable SAT Solver Challenge[2]). Therefore, algorithm selection can be applied to select a well-performing configurator for a given configuration scenario. Bischl et al. (2012) proposed algorithm selection for black-box optimization; however, they used only synthetic black-box functions and have not applied it to algorithm configuration. Alternatively, schedules of configurators could be used and interact with another to speed up the configuration process. As also already mentioned, different algorithm selection strategies perform well on different scenarios. Additionally, algorithm selectors are typically based on machine learning methods having parameters itself. So, algorithm configuration can be also applied to selectors, such as *claspfolio* 2, to improve their performance for a certain scenario. Related to this, Thornton et al. (2013) proposed *AutoWeka*, an algorithm configuration framework to automatically find a well-performing machine learning approach with its hyper-parameters for a given data set. This relates to algorithm configuration for selectors, since selectors typically also rely on machine learning approaches. However, all other parts of algorithm selection, such as pre-solving schedules, are not yet considered in algorithm configuration scenarios.

8.3 Thesis Contributions in a Nutshell

The meta-algorithmic methods we have introduced in this work, namely ACPP, *aspeed*, *claspfolio* 2 and *PASU*, turned out to perform well, especially on heterogeneous instance sets. We can automatically improve the robustness of algorithms, in particular, solvers for NP-hard problems; using them reduces the need of human experts to select and configure solvers for new applications. In the light of increasing importance of parallel processor technology and high performance computation, we extended all methods to be applicable to parallel solving.

[2] http://aclib.net/cssc2014/

List of Figures

List of Tables

List of Algorithms

A Notation

Algorithm (e.g., solvers)	$a \in A$
Alignment	$\pi : \{1, \ldots, \lvert A \rvert\} \to A$
Cluster	$s \in S$
Cluster Mapping	$s : I \to S$
Configuration	$c \in C$
Configuration Budget	t_b
Configurator	AC
Features	$f : I \to \mathbb{R}^d$
Hardness Metric	$h : I \to \mathbb{R}$
Instance	$i \in I$
Performance Metric	$m : I \times A \to \mathbb{R}$
Portfolio of k Configurations	$c_{1:k}$
Portfolio Size	$k \in \mathbb{R}$
Q-score	$I \times I \times A \times m \to \mathbb{R}$
Runtimes	$t : I \times A \to \mathbb{R}$
Runtime Cutoff	t_c
Schedule	$\sigma : A \to [0, t_c]$
Selection Mapping	$\phi : I \to A$
Threshold	$e \in \mathbb{R}$

Table A.1: Overview of Notation

B *claspfolio* 2 on ASlib

All listed results include unsolvable instances in contrast to Figure 5.7.

	#Timeouts	PAR10	PAR1
3S-like	102	501.79	76.13
aspeed	149	774.72	152.93
claspfolio-1.0-like	109	536.65	81.78
ISAC-like	107	525.78	79.25
ME-ASP-like	134	649.95	90.75
SATzilla'09-like	115	561.14	81.23
SATzilla'11-like	101	497.15	75.66
single best	183	880.55	116.87

Table B.1: *ASP-POTASSCO*: 11 algorithms, oracle (par10): 400.17, feature costs: 1.32

	#Timeouts	PAR10	PAR1
3S-like	273	6901.28	831.61
aspeed	277	7079.08	920.48
claspfolio-1.0-like	271	6775.77	750.57
ISAC-like	268	6708.35	749.85
ME-ASP-like	264	6609.21	739.65
SATzilla'09-like	264	6612.85	743.29
SATzilla'11-like	263	6580.71	733.38
single best	288	7201.56	798.39

Table B.2: *CSP-2010*: 2 algorithms, oracle (par10): 6344.25, feature costs: 0.00

	#Timeouts	PAR10	PAR1
3S-like	139	3387.90	388.93
aspeed	136	3749.81	815.57
claspfolio-1.0-like	167	4033.44	430.36
ISAC-like	144	3487.49	380.64
ME-ASP-like	164	3969.84	431.48
SATzilla'09-like	149	3618.06	403.34
SATzilla'11-like	138	3343.59	366.20
single best	202	4893.14	534.92

Table B.3: *MAXSAT12-PMS*: 6 algorithms, oracle (par10): 3127.23, feature costs: 0.15

	#Timeouts	PAR10	PAR1
3S-like	27	2122.35	462.39
aspeed	25	2165.90	628.90
claspfolio-1.0-like	86	6073.31	786.02
ISAC-like	75	5353.12	742.12
ME-ASP-like	91	6431.63	836.94
SATzilla'09-like	63	4524.39	651.14
SATzilla'11-like	54	3921.90	601.97
single best	99	7002.91	916.38

Table B.4: *PREMARSHALLING-ASTAR-2013*: 4 algorithms, oracle (par10): 227.60, feature costs: 0.00

	#Timeouts	PAR10	PAR1
3S-like	344	9237.10	1089.73
aspeed	349	9853.57	1587.78
claspfolio-1.0-like	430	11406.75	1222.54
ISAC-like	408	10838.02	1174.86
ME-ASP-like	405	10756.89	1164.79
SATzilla'09-like	380	10083.66	1083.66
SATzilla'11-like	340	9063.60	1010.96
single best	579	15330.17	1617.01

Table B.5: *QBF-2011*: 5 algorithms, oracle (par10): 8337.09, feature costs: 0.00

	#Timeouts	PAR10	PAR1
3S-like	106	18307.00	2192.13
aspeed	100	17497.90	2295.20
claspfolio-1.0-like	141	24276.22	2840.41
ISAC-like	141	24270.92	2835.10
ME-ASP-like	153	26297.90	3037.77
SATzilla'09-like	113	19553.06	2374.01
SATzilla'11-like	116	19996.89	2361.75
single best	152	26188.09	3079.98

Table B.6: *SAT11-HAND*: 15 algorithms, oracle (par10): 13360.66, feature costs: 41.22

	#Timeouts	PAR10	PAR1
3S-like	83	14369.00	1919.00
aspeed	89	15379.63	2029.63
claspfolio-1.0-like	82	14121.53	1821.53
ISAC-like	79	13618.07	1768.07
ME-ASP-like	82	14147.65	1847.65
SATzilla'09-like	84	14457.12	1857.12
SATzilla'11-like	82	14129.46	1829.46
single best	90	15411.24	1911.24

Table B.7: *SAT11-INDU*: 18 algorithms, oracle (par10): 8187.51, feature costs: 135.34

	#Timeouts	PAR10	PAR1
3S-like	143	12151.33	1426.33
aspeed	133	11631.68	1656.68
claspfolio-1.0-like	170	14310.28	1560.28
ISAC-like	166	13973.82	1523.82
ME-ASP-like	187	15701.03	1676.03
SATzilla'09-like	164	13791.19	1491.19
SATzilla'11-like	154	12982.65	1432.65
single best	254	21249.66	2199.66

Table B.8: *SAT11-RAND*: 8 algorithms, oracle (par10): 9186.44, feature costs: 22.06

	#Timeouts	PAR10	PAR1
3S-like	264	2113.40	346.85
aspeed	357	2964.27	575.42
claspfolio-1.0-like	330	2572.93	364.76
ISAC-like	370	2857.17	381.33
ME-ASP-like	381	2938.29	388.85
SATzilla'09-like	290	2275.44	334.92
SATzilla'11-like	249	1969.20	303.03
single best	399	3079.89	410.00

Table B.9: *SAT12-ALL*: 31 algorithms, oracle (par10): 241.31, feature costs: 40.58

	#Timeouts	PAR10	PAR1
3S-like	328	5224.27	605.75
aspeed	332	5374.63	699.79
claspfolio-1.0-like	378	5979.42	656.87
ISAC-like	389	6159.70	682.26
ME-ASP-like	401	6343.97	697.56
SATzilla'09-like	343	5439.99	610.27
SATzilla'11-like	339	5375.73	602.33
single best	401	6338.90	692.49

Table B.10: *SAT12-HAND*: 31 algorithms, oracle (par10): 3662.24, feature costs: 39.06

	#Timeouts	PAR10	PAR1
3S-like	286	3082.68	435.90
aspeed	325	3548.74	541.03
claspfolio-1.0-like	290	3104.45	420.64
ISAC-like	284	3042.80	414.53
ME-ASP-like	295	3157.45	427.37
SATzilla'09-like	288	3086.89	421.60
SATzilla'11-like	282	3017.90	408.14
single best	308	3266.05	415.66

Table B.11: *SAT12-INDU*: 31 algorithms, oracle (par10): 2221.49, feature costs: 80.90

	#Timeouts	PAR10	PAR1
3S-like	374	3353.19	387.55
aspeed	369	3343.20	417.21
claspfolio-1.0-like	369	3302.62	376.62
ISAC-like	370	3308.98	375.06
ME-ASP-like	372	3328.74	378.96
SATzilla'09-like	370	3312.28	378.36
SATzilla'11-like	370	3309.19	375.27
single best	366	3271.14	368.93

Table B.12: *SAT12-RAND*: 31 algorithms, oracle (par10): 2872.84, feature costs: 9.02

C Portfolio of *clasp* Configurations for *RICOCHET ROBOTS* and *ASP-POTASSCO*

C.1 *RICOCHET ROBOTS*

```
--eq=0 --trans-ext=no --sat-prepro=0 --update-lbd=2 --heuristic=Vsids --sign-def=1
--restart-on-model --opt-heuristic=3 --vsids-decay=81 --local-restarts --lookahead=no
--otfs=0 --reverse-arcs=0 --save-progress=0 --init-watches=2 --restarts=D,240,0.513,127
--opt-hierarch=3 --strengthen=recursive,0 --deletion=0 --loops=common --del-on-restart=0
--contraction=no

--backprop --eq-dfs --eq=8 --trans-ext=dynamic --sat-prepro=0 --sign-def=1
--restart-on-model --strengthen=recursive,0 --del-init-r=1,8594 --loops=no
--del-max=453160547 --reverse-arcs=3 --heuristic=Vsids --restarts=x,1734,1.9172,477
--deletion=0 --update-act --contraction=no --update-lbd=1 --opt-heuristic=0
--vsids-decay=94 --otfs=1 --init-moms --del-on-restart=50 --init-watches=2
--local-restarts --lookahead=no --save-progress=0 --opt-hierarch=0 --sign-fix

--backprop --eq=1 --trans-ext=no --sat-prepro=0 --sign-def=1 --restart-on-model
--strengthen=local,1 --init-watches=0 --del-init-r=25,20790 --loops=shared
--del-max=1312983571 --reverse-arcs=0 --heuristic=Vsids
--restarts=x,519,1.788,5 --del-algo=inp_sort,0 --del-estimate
--del-grow=1.5672,31.7771,+,137,218 --update-act --del-glue=3,0 --update-lbd=2
--opt-heuristic=1 --deletion=1,90,9.2504 --vsids-decay=70 --otfs=2 --init-moms
--del-on-restart=0 --contraction=no --lookahead=no --save-progress=0 --opt-hierarch=3

--backprop --eq=37 --trans-ext=dynamic --sat-prepro=0 --sign-def=1 --restart-on-model
--strengthen=local,1 --init-watches=2 --del-init-r=56,15164 --loops=shared
--del-max=511359929 --reverse-arcs=3 --heuristic=Vsids --del-cfl=L,1 --restarts=no
--del-algo=inp_sort,0 --deletion=2,36,1.8454 --del-glue=7,0 --update-lbd=0
--opt-heuristic=3 --del-estimate --vsids-decay=82 --otfs=0 --del-on-restart=0
--contraction=no --local-restarts --lookahead=no --save-progress=180 --opt-hierarch=3
--sign-fix

--backprop --eq-dfs --eq=1 --trans-ext=weight --sat-prepro=0 --update-lbd=0
--heuristic=Berkmin --sign-def=1 --init-moms --opt-heuristic=1 --strengthen=local,2
--lookahead=no --reverse-arcs=2 --save-progress=61 --restarts=no --otfs=2
--opt-hierarch=2 --init-watches=2 --deletion=0 --berk-max=3 --loops=no
--update-act --del-on-restart=41 --sign-fix --contraction=1
```

```
(Single Best) --eq=1 --trans-ext=dynamic --sat-prepro=3,41,-1,45,1 --update-lbd=0
--heuristic=Vsids --sign-def=2 --del-max=2053365695 --opt-heuristic=3
--vsids-decay=84 --strengthen=recursive,0 --lookahead=hybrid,14 --reverse-arcs=0
--save-progress=180 --del-init-r=24,5723 --restarts=+,1,1 --otfs=1
--opt-hierarch=2 --init-watches=0 --deletion=0 --loops=no --update-act
--del-on-restart=50 --init-moms --contraction=no

--eq-dfs --eq=1 --trans-ext=dynamic --sat-prepro=0 --sign-def=0
--restart-on-model --strengthen=local,2 --loops=no --init-watches=2
--heuristic=Vsids --reverse-arcs=1 --del-cfl=F,14080 --restarts=D,420,0.5333,37
--del-algo=basic,1 --deletion=2,61,9.0412 --update-act --del-glue=7,1
--update-lbd=3 --opt-heuristic=3 --vsids-decay=92 --otfs=0 --del-on-restart=4
--contraction=no --local-restarts --lookahead=no --save-progress=19
--opt-hierarch=2

--eq=0 --trans-ext=integ --sat-prepro=0 --sign-def=1 --del-max=1679545344
--strengthen=recursive,1 --init-watches=0 --del-init-r=89,8153 --loops=no
--restart-on-model --reverse-arcs=2 --heuristic=Vsids --restarts=x,3,1.9488
--del-algo=sort,0 --deletion=1,49,8.3325 --del-grow=1.2595,73.1424,x,4,1.8635
--update-act --del-glue=1,1 --update-lbd=2 --opt-heuristic=0 --vsids-decay=89
--otfs=0 --init-moms --del-on-restart=20 --contraction=no --lookahead=no --save-progress=64
--opt-hierarch=1 --sign-fix

--eq=0 --trans-ext=dynamic --sat-prepro=0 --sign-def=1 --del-max=1908872295
--strengthen=local,1 --del-init-r=3,7089 --loops=distinct --init-watches=2
--heuristic=None --reverse-arcs=1 --restarts=no --del-algo=inp_heap,1
--del-estimate --del-grow=4.3442,4.3608,+,1,11136,1 --del-glue=7,1
--update-lbd=0 --opt-heuristic=2 --deletion=1,100,8.5016 --otfs=2
--init-moms --del-on-restart=0 --contraction=no --lookahead=no
--save-progress=11 --opt-hierarch=1

--eq=53 --trans-ext=all --sat-prepro=0 --update-lbd=1 --heuristic=Vsids
--del-on-restart=50 --sign-def=2 --opt-heuristic=3 --vsids-decay=78
--strengthen=local,0 --lookahead=no --reverse-arcs=2 --save-progress=180
--del-cfl=F,31799 --restarts=no --otfs=0 --del-algo=inp_sort,2 --init-watches=2
--deletion=2,8,4.3982 --contraction=8 --loops=no --opt-hierarch=3 --del-glue=0,0

--backprop --eq=0 --trans-ext=all --sat-prepro=0 --sign-def=0 --strengthen=local,0
--loops=distinct --init-watches=2 --heuristic=Vsids --reverse-arcs=3
--restarts=D,304,0.7808,109 --del-algo=inp_heap,0 --del-estimate
--del-grow=2.4788,76.3835,L,448,3758 --update-act --del-glue=6,0
--update-lbd=0 --opt-heuristic=0 --deletion=1,48,6.3707 --vsids-decay=87
--otfs=1 --init-moms --del-on-restart=25 --contraction=no --lookahead=no
--save-progress=171 --opt-hierarch=0
```

```
--eq=0 --trans-ext=choice --sat-prepro=0 --sign-def=2 --restart-on-model
--strengthen=recursive,1 --loops=no --reverse-arcs=0 --heuristic=Vsids
--restarts=no --del-algo=inp_sort,0 --deletion=1,2,3.0926
--del-grow=1.6048,50.6736,F,7 --update-act --contraction=no
--del-glue=6,0 --update-lbd=1 --opt-heuristic=3 --vsids-decay=75
--otfs=1 --init-moms --del-on-restart=0 --init-watches=2
--local-restarts --lookahead=no --save-progress=0 --opt-hierarch=2

--backprop --eq=0 --trans-ext=no --sat-prepro=0 --sign-def=2
--del-max=100521649 --strengthen=recursive,2 --init-watches=0
--del-init-r=535,3187 --loops=shared --reverse-arcs=3 --heuristic=Berkmin
--berk-once --del-cfl=F,7 --restarts=no --del-algo=sort,1 --del-estimate
--berk-max=326 --del-grow=3.3683,29.9185,L,1,804 --del-glue=0,0
--update-lbd=2 --opt-heuristic=0 --deletion=3,100,1.7264 --otfs=0
--del-on-restart=0 --contraction=68 --local-restarts --lookahead=no
--save-progress=0 --opt-hierarch=1

--eq-dfs --eq=3 --trans-ext=choice --sat-prepro=1,41,-1,27,2 --sign-def=1
--del-max=400721214 --strengthen=recursive,0 --init-watches=2
--del-init-r=539,2196 --loops=shared --reverse-arcs=0
--heuristic=Vsids --restarts=L,2 --del-algo=inp_heap,0
--deletion=1,69,9.371 --del-grow=2.6076,85.1081,F,185 --update-act
--del-glue=1,1 --update-lbd=1 --opt-heuristic=3 --vsids-decay=79
--otfs=1 --del-on-restart=36 --contraction=376 --counter-restarts=73
--lookahead=no --save-progress=93 --opt-hierarch=2 --counter-bump=164

--eq=0 --trans-ext=integ --sat-prepro=0 --sign-def=1 --restart-on-model
--strengthen=local,2 --loops=distinct --init-watches=2 --heuristic=Vsids
--reverse-arcs=0 --del-cfl=L,1067 --restarts=+,212,23685,135
--del-algo=basic,1 --deletion=2,95,1.6916 --del-glue=0,1 --update-lbd=3
--opt-heuristic=2 --vsids-decay=89 --otfs=0 --del-on-restart=50
--contraction=52 --local-restarts --lookahead=no --save-progress=21
--opt-hierarch=1
```

C.2 *ASP-POTASSCO*

```
--eq-dfs --eq=125 --trans-ext=dynamic --sat-prepro=0 --update-lbd=3
--heuristic=Vsids --sign-def=0 --opt-heuristic=0 --vsids-decay=92
--strengthen=recursive,2 --lookahead=no --otfs=1 --reverse-arcs=3
--save-progress=7 --restarts=L,14,5 --opt-hierarch=2 --init-watches=0
--deletion=0 --loops=no --update-act --del-on-restart=8 --sign-fix
 --contraction=3
```

```
--backprop --eq=0 --trans-ext=dynamic --sat-prepro=10,25,-1,100,1
--sign-def=1 --del-max=32767 --strengthen=local,0 --init-watches=2
--del-init-r=1000,9000 --loops=no --reverse-arcs=1 --heuristic=Vsids
--del-cfl=+,10000,1000 --restarts=x,128,1.5 --del-algo=basic,0
--deletion=3,75,10.0 --del-grow=1.1,20.0,L,1000 --del-glue=2,0
--update-lbd=0 --opt-heuristic=0 --vsids-decay=70 --otfs=2
--del-on-restart=30 --contraction=no --counter-restarts=3
--lookahead=no --save-progress=180 --opt-hierarch=0 --counter-bump=10

--eq=75 --trans-ext=all --sat-prepro=30,5,-1,22,2 --sign-def=0
--restart-on-model --strengthen=recursive,2 --init-watches=0
--loops=distinct --reverse-arcs=0 --heuristic=Berkmin
--del-cfl=x,192,1.499 --restarts=+,98,281,642 --del-algo=sort,1
--deletion=3,63,5.5002 --berk-max=1 --del-grow=4.3241,25.8048,x,2023,1.3915
--update-act --del-glue=3,0 --update-lbd=0 --opt-heuristic=3 --otfs=1
--init-moms --del-on-restart=36 --contraction=no --lookahead=atom,106
--save-progress=101 --opt-hierarch=0

--eq=1 --trans-ext=all --sat-prepro=0 --sign-def=2 --restart-on-model
--strengthen=recursive,1 --init-watches=2 --del-init-r=824,10792
--loops=shared --del-max=211572149 --reverse-arcs=2 --heuristic=Vsids
--del-cfl=+,96,250 --restarts=+,3594,10909,9 --del-algo=inp_heap,2
--del-estimate --update-act --del-glue=4,0 --update-lbd=1 --opt-heuristic=2
--deletion=2,87,4.7191 --vsids-decay=90 --otfs=2 --del-on-restart=37
--contraction=no --counter-restarts=2 --local-restarts --lookahead=no
--save-progress=98 --opt-hierarch=2 --counter-bump=65

--backprop --eq=0 --trans-ext=card --sat-prepro=6,2,-1,85,0 --sign-def=1
--del-max=1821250312 --strengthen=local,2 --del-init-r=237,15629
--loops=common --init-watches=2 --heuristic=Vsids --reverse-arcs=1
--restarts=+,13,56 --deletion=0 --update-lbd=2 --opt-heuristic=2
--vsids-decay=76 --otfs=1 --del-on-restart=13 --contraction=no
--counter-restarts=2 --lookahead=atom,2 --save-progress=166
--opt-hierarch=1 --counter-bump=201 --sign-fix

--backprop --eq=0 --trans-ext=choice --sat-prepro=10,25,-1,100,1
--sign-def=1 --del-max=32767 --strengthen=recursive,0 --del-init-r=1000,9000
--loops=no --init-watches=2 --heuristic=Vsids --reverse-arcs=0
--del-cfl=+,10000,1000 --restarts=L,128 --del-algo=basic,0
--deletion=3,75,10.0 --del-grow=1.1,20.0,L,1000 --del-glue=2,0
--update-lbd=0 --opt-heuristic=3 --vsids-decay=70 --otfs=1 --del-on-restart=37
--contraction=no --lookahead=no --save-progress=180 --opt-hierarch=0

--eq=58 --trans-ext=all --sat-prepro=0 --sign-def=1 --strengthen=local,0
```

```
--init-watches=2 --loops=shared --reverse-arcs=1 --heuristic=Vsids
--del-cfl=F,4048 --restarts=x,2768,1.2981 --del-algo=inp_sort,0
--deletion=3,93,9.7528 --del-grow=4.214,63.6462,+,1,571,2461 --del-glue=6,1
--update-lbd=3 --opt-heuristic=2 --vsids-decay=71 --otfs=0 --init-moms
--del-on-restart=47 --contraction=no --local-restarts --lookahead=no
--save-progress=42 --opt-hierarch=3 --sign-fix

--backprop --eq-dfs --eq=7 --trans-ext=all --sat-prepro=2,14,-1,29,0
--sign-def=0 --del-max=435043091 --strengthen=recursive,0 --init-watches=0
--del-init-r=59,26445 --loops=no --reverse-arcs=3 --heuristic=Vsids
--del-cfl=x,5,1.9291,12 --restarts=x,1,1.0611 --del-algo=inp_heap,1
--deletion=3,10,9.3132 --del-grow=1.3879,19.738,F,445 --del-glue=5,1
--update-lbd=0 --opt-heuristic=2 --vsids-decay=93 --otfs=2 --del-on-restart=36
--contraction=no --lookahead=hybrid,3 --save-progress=175 --opt-hierarch=0

--eq-dfs --eq=1 --trans-ext=choice --sat-prepro=0 --sign-def=2
--restart-on-model --strengthen=local,0 --del-init-r=4,13882 --loops=no
--del-max=201830159 --init-watches=0 --heuristic=Berkmin --berk-once
--reverse-arcs=1 --restarts=F,5109 --deletion=0 --berk-max=7 --update-act
--update-lbd=3 --opt-heuristic=0 --otfs=0 --berk-huang --del-on-restart=4
--contraction=78 --counter-restarts=30 --lookahead=atom,2 --save-progress=37
--opt-hierarch=2 --counter-bump=214

--backprop --eq=5 --trans-ext=integ --sat-prepro=10,25,-1,100,1 --update-lbd=0
--opt-heuristic=0 --sign-def=1 --del-algo=basic,0 --strengthen=local,0 --reverse-arcs=1
--vsids-decay=70 --save-progress=180 --contraction=no --restarts=x,128,1.0582
--otfs=2
--opt-hierarch=0 --init-watches=2 --heuristic=Vsids --deletion=1,75,10.0 --lookahead=no
--loops=common --del-grow=1.1,20.0,L,1000 --update-act --del-on-restart=30 --del-glue=2,0

--eq-dfs --eq=1 --trans-ext=no --sat-prepro=4,36,-1,75,0 --sign-def=1
--restart-on-model --strengthen=recursive,1 --init-watches=2 --loops=shared
--reverse-arcs=1 --heuristic=Berkmin --berk-once --restarts=D,58,0.8882,31
--del-algo=sort,1 --del-estimate --berk-max=4 --del-grow=3.2967,91.2359,L,26294,221
--update-act --del-glue=3,1 --update-lbd=3 --opt-heuristic=1 --deletion=1,36,1.7415
--otfs=0 --del-on-restart=35 --contraction=no --counter-restarts=67 --local-restarts
--lookahead=no --save-progress=73 --opt-hierarch=2 --counter-bump=1100 --sign-fix
```

Bibliography

Aigner, M., Biere, A., Kirsch, C., Niemetz, A., & Preiner, M. (2013). Analysis of portfolio-style parallel SAT solving on current multi-core architectures. In *Proceeding of the Fourth International Workshop on Pragmatics of SAT (POS'13)*.

Alviano, M., Dodaro, C., Faber, W., Leone, N., & Ricca, F. (2013). WASP: A native asp solver based on constraint learning.. In Cabalar, & Son (Cabalar & Son, 2013), pp. 54–66.

Amadini, R., Gabbrielli, M., & Mauro, J. (2014). Portfolio approaches for constraint optimization problems. In *Proceedings of the Conference on Learning and Intelligent OptimizatioN (LION'14)*, pp. 21–35.

Andres, B., Kaufmann, B., Matheis, O., & Schaub, T. (2012). Unsatisfiability-based optimization in clasp.. In Dovier, & Santos Costa (Dovier & Santos Costa, 2012), pp. 212–221.

Ansótegui, C., Sellmann, M., & Tierney, K. (2009). A gender-based genetic algorithm for the automatic configuration of algorithms. In Gent, I. (Ed.), *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP'09)*, Vol. 5732 of *Lecture Notes in Computer Science*, pp. 142–157. Springer-Verlag.

Asin, R., Olate, J., & Ferres, L. (2013). Cache performance study of portfolio-based parallel CDCL SAT solvers. *CoRR*, *abs/1309.3187*.

Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.-M., & Piette, C. (2012). Penelope, a parallel clause-freezer solver.. In Balint et al. (Balint, Belov, Diepold, Gerber, Järvisalo, & Sinz, 2012a), pp. 43–44. Available at `https://helda.helsinki.fi/handle/10138/34218`.

Audemard, G., & Simon, L. (2012). Glucose 2.1. in the SAT challenge 2012.. In Balint et al. (Balint et al., 2012a), pp. 23–23. Available at `https://helda.helsinki.fi/handle/10138/34218`.

Balint, A., Belov, A., Diepold, D., Gerber, S., Järvisalo, M., & Sinz, C. (Eds.). (2012a). *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, Vol. B-2012-2 of *Department of Computer Science Series of Publications B*. University of Helsinki. Available at `https://helda.helsinki.fi/handle/10138/34218`.

Balint, A., Belov, A., Järvisalo, M., & Sinz, C. (2012b). Application and hard combinatorial benchmarks in SAT challenge 2012.. In Balint et al. (Balint et al., 2012a), pp. 69–71. Available at `https://helda.helsinki.fi/handle/10138/34218`.

Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Bayless, S., Tompkins, D., & Hoos, H. (2012). Evaluating instance generators by configuration. Tech. rep., Department of Computer Science, University of British Columbia.

Berre, D., Roussel, O., & Simon, L. (2009). `http://www.satcompetition.org/2009/BenchmarksSelection.html`, last visited. 09-03-2012.

Bessiere, C. (Ed.). (2007). *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07)*, Vol. 4741 of *Lecture Notes in Computer Science*. Springer-Verlag.

Biere, A. (2010). Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. Tech. rep. 10/1, Institute for Formal Models and Verification. Johannes Kepler University.

Biere, A. (2011). Lingeling and friends at the SAT competition 2011. Technical report FMV 11/1, Institute for Formal Models and Verification, Johannes Kepler University.

Biere, A. (2012). Lingeling and friends entering the SAT challenge 2012.. In Balint et al. (Balint et al., 2012a), pp. 33–34. Available at `https://helda.helsinki.fi/handle/10138/34218`.

Biere, A. (2013). Lingeling, plingeling and treengeling entering the sat competition 2013. In Balint, A., Belov, A., Heule, M., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, Vol. B-2013-1 of *Department of Computer Science Series of Publications B*, pp. 51–52. University of Helsinki.

Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.). (2009). *Handbook of Satisfiability*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Bischl, B., Mersmann, O., Trautmann, H., & Preuß, M. (2012). Algorithm selection based on exploratory landscape analysis and cost-sensitive learning. In Soule, T., & Moore, J. (Eds.), *Proceedings of the International Conference on Genetic and Evolutionary Computation (GECCO'12)*, pp. 313–320. ACM.

Bishop, C. (2007). *Pattern Recognition and Machine Learning (Information Science and Statistics)* (2 edition). Springer-Verlag.

Boutilier, C. (Ed.). (2009). *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*. AAAI/MIT Press.

Brglez, F., Li, X., & Stallmann, F. (2002). The role of a skeptic agent in testing and benchmarking of sat algorithms..

Cabalar, P., & Son, T. (Eds.). (2013). *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, Vol. 8148 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Cai, S., Luo, C., & Su, K. (2012). Ccasat: Solver description.. In Balint et al. (Balint et al., 2012a), pp. 13–14. Available at `https://helda.helsinki.fi/handle/10138/34218`.

Calimeri, F., Ianni, G., & Ricca, F. (2011a). Third ASP competition - file and language formats. Tech. rep., Università della Calabria.

Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M., Sirianni, M., Terracina, G., & Veltri, P. (2011b). The third answer set programming competition: Preliminary report of the system competition track.. In Delgrande, & Faber (Delgrande & Faber, 2011), pp. 388–403.

Cavazos, J., & O'Boyle, M. (2005). Automatic tuning of inlining heuristics. In Kramer, W. (Ed.), *Proceedings of the International Conference on High Performance Networking and Computing*, pp. 1–14. IEEE Computer Society.

Chen, J. (2011). Phase selection heuristics for satisfiability solvers. *CoRR*, *abs/1106.1372*.

Cimatti, A., & Sebastiani, R. (Eds.). (2012). *Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, Vol. 7317 of *Lecture Notes in Computer Science*. Springer-Verlag.

Coelho, H., Studer, R., & Wooldridge, M. (Eds.). (2010). *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10)*. IOS Press.

Collautti, M., Malitsky, Y., Mehta, D., & O'Sullivan, B. (2013). SNAPP: Solver-based nearest neighbor for algorithm portfolios. In Zelezny, F. (Ed.), *Proceedings of the Twenty-Fourth European Conference on Machine Learning (ECML'13)*, Lecture Notes in Computer Science. Springer-Verlag.

Delgrande, J., & Faber, W. (Eds.). (2011). *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, Vol. 6645 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Diao, Y., Eskesen, F., Froehlich, S., Hellerstein, J. L., Spainhower, L., & Surendra, M. (2003). Generic online optimization of multiple configuration parameters with application to a database server. In Brunner, M., & Keller, A. (Eds.), *Proceedings of the Fourteenth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'03)*, Vol. 2867 of *Lecture Notes in Computer Science*, pp. 3–15. Springer-Verlag.

Dovier, A., & Santos Costa, V. (Eds.). (2012). *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12)*, Vol. 17. Leibniz International Proceedings in Informatics (LIPIcs).

Eén, N., & Sörensson, N. (2004). An extensible SAT-solver. In Giunchiglia, E., & Tacchella, A. (Eds.), *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, Vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer-Verlag.

Gagliolo, M., & Schmidhuber, J. (2006). Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, *47*(3-4), 295–328.

Gebruers, C., Guerri, A., Hnich, B., & Milano, M. (2004). Making choices using structure at the instance level within a case based reasoning framework. In Régin, J., & Rueher, M. (Eds.), *Proceedings of the First Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'04)*, Vol. 3011 of *Lecture Notes in Computer Science*, pp. 380–386. Springer-Verlag.

Gebser, M., Jost, H., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T., & Schneider, M. (2013). Ricochet robots: A transverse ASP benchmark.. In Cabalar, & Son (Cabalar & Son, 2013), pp. 348–360.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., & Schneider, M. (2011). Potassco: The Potsdam answer set solving collection. *AI Communications*, *24*(2), 107–124.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., & Thiele, S. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`..

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2012). *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M., & Ziller, S. (2011). A portfolio solver for answer set programming: Preliminary report.. In Delgrande, & Faber (Delgrande & Faber, 2011), pp. 352–357.

Gebser, M., Kaminski, R., & Schaub, T. (2012a). Gearing up for effective ASP planning. In Erdem, E., Lee, J., Lierler, Y., & Pearce, D. (Eds.), *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, Vol. 7265 of *Lecture Notes in Computer Science*, pp. 296–310. Springer-Verlag.

Gebser, M., Kaufmann, B., & Schaub, T. (2012b). Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, *187-188*, 52–89.

Gebser, M., Kaufmann, B., & Schaub, T. (2012c). Multi-threaded ASP solving with clasp. *Theory and Practice of Logic Programming*, *12*(4-5), 525–545.

Gent, I., Jefferson, C., Kotthoff, L., Miguel, I., Moore, N., Nightingale, P., & Petrie, K. (2010). Learning when to use lazy learning in constraint solving.. In Coelho et al. (Coelho, Studer, & Wooldridge, 2010), pp. 873–878.

Giunchiglia, E., Lierler, Y., & Maratea, M. (2006). Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, *36*(4), 345–377.

Gomes, C., & Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, *126*(1-2), 43–62.

Grinten, A., Wotzlaw, A., Speckenmeyer, E., & Porschen, S. (2012). satUZK: Solver description.. In Balint et al. (Balint et al., 2012a), pp. 54–55. Available at `https://helda.helsinki.fi/handle/10138/34218`.

Guerri, A., & Milano, M. (2004). Learning techniques for automatic algorithm portfolio selection. In López de Mántaras, R., & Saitta, L. (Eds.), *Proceedings of the Sixteenth Eureopean Conference on Artificial Intelligence (ECAI'04)*, pp. 475–479. IOS Press.

Guo, H., & Hsu, W. (2004). A learning-based algorithm selection meta-reasoner for the real-time MPE problem. In *Proceedings of the Seventeenth Australian Joint Conference on Artificial Intelligence*, pp. 307–318. Springer-Verlag.

Guo, L., Hamadi, Y., Jabbour, S., & Sais, L. (2010). Diversification and intensification in parallel SAT solving. In Cohen, D. (Ed.), *Proceedings of the Sixteenth International Conference on Principles and Practice of Constraint Programming (CP'10)*, Vol. 6308 of *Lecture Notes in Computer Science*, pp. 252–265. Springer-Verlag.

Hamadi, Y., Jabbour, S., Piette, C., & Sais, L. (2011). Deterministic parallel DPLL. *Journal on Satisfiability, Boolean Modeling and Computation*, *7*(4), 127–132.

Hamadi, Y., Jabbour, S., & Sais, L. (2009a). Control-based clause sharing in parallel SAT solving.. In Boutilier (Boutilier, 2009), pp. 499–504.

Hamadi, Y., Jabbour, S., & Sais, L. (2009b). ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, *6*, 245–262.

Hamadi, Y., & Schoenauer, M. (Eds.). (2012). *Proceedings of the Sixth International Conference Learning and Intelligent Optimization (LION'12)*, Vol. 7219 of *Lecture Notes in Computer Science*. Springer-Verlag.

Hamadi, Y., & Wintersteiger, C. M. (2012). Seven challenges in parallel SAT solving. In Hoffmann, J., & Selman, B. (Eds.), *Proceedings of the Twenty-Sixth National Conference on Artificial Intelligence (AAAI'12)*. AAAI Press.

Hamerly, G., & Elkan, C. (2003). Learning the k in k-means. In Thrun, S., Saul, L., & Schölkopf, B. (Eds.), *Proceedings of the Sixteenth International Conference on Advances in Neural Information Processing Systems (NIPS'03)*. MIT Press.

Helmert, M., Röger, G., & Karpas, E. (2011). Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS 2011 Workshop on Planning and Learning*, pp. 28–35.

Heule, M., Dufour, M., van Zwieten, J., & van Maaren, H. (2004). March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In Hoos, H., & Mitchell, D. (Eds.), *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Vol. 3542 of *Lecture Notes in Computer Science*, pp. 345–359. Springer-Verlag.

Hill, T., & Lewicki, P. (2005). *Statistics: Methods and Applications*. StatSoft.

Holte, R. C., & Howe, A. (Eds.). (2007). *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07)*. AAAI Press.

Hoos, H. (2012). Programming by optimisation. *Communications of the ACM*, *55*, 70–80.

Hoos, H., Kaminski, R., Lindauer, M., & Schaub, T. (2015). aspeed: Solver scheduling via answer set programming. *Theory and Practice of Logic Programming*, *15*, 117–142.

Hoos, H., Kaminski, R., Schaub, T., & Schneider, M. (2012). aspeed: ASP-based solver scheduling.. In Dovier, & Santos Costa (Dovier & Santos Costa, 2012), pp. 176–187.

Hoos, H., Kaufmann, B., Schaub, T., & Schneider, M. (2013). Robust benchmark set selection for boolean constraint solvers.. In Pardalos, & Nicosia (Pardalos & Nicosia, 2013), pp. 138–152.

Hoos, H., Leyton-Brown, K., Schaub, T., & Schneider, M. (2012). Algorithm configuration for portfolio-based parallel SAT-solving. In Coletta, R., Guns, T., O'Sullivan, B., Passerini, A., & Tack, G. (Eds.), *Proceedings of the First Workshop on Combining Constraint Solving with Mining and Learning (CoCoMile'12)*, pp. 7–12.

Hoos, H., Lindauer, M., & Schaub, T. (2014). claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming*, *14*(4-5), 569–585.

Hoos, H., & Stützle, T. (2004). *Stochastic Local Search: Foundations and Applications*. Elsevier/-Morgan Kaufmann.

Huberman, B., Lukose, R., & Hogg, T. (1997). An economic approach to hard computational problems. *Science*, *275*, 51–54.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2010). Automated configuration of mixed integer programming solvers. In *Proceedings of the Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming (CPAIOR)*, pp. 186–202.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2011a). Sequential model-based optimization for general algorithm configuration. In *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization (LION'11)*, Vol. 6683 of *Lecture Notes in Computer Science*, pp. 507–523. Springer-Verlag.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2011b). Tradeoffs in the empirical evaluation of competing algorithm designs. *Annals of Mathematics and Artificial Intelligence*, *60*(1), 65–89.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2014). Submodular configuration of algorithms for portfolio-based selection. Tech. rep., Department of Computer Science, University of British Columbia. (to appear).

Hutter, F., Hoos, H., Leyton-Brown, K., & Stützle, T. (2009). ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, *36*, 267–306.

Hutter, F., Hoos, H., & Stützle, T. (2007). Automatic algorithm configuration based on local search.. In Holte, & Howe (Holte & Howe, 2007), pp. 1152–1157.

Hutter, F., Lopez-Ibanez, M., Fawcett, C., Lindauer, M., Hoos, H., Leyton-Brown, K., & Stützle, T. (2014a). AClib: A benchmark library for algorithm configuration. In Pardalos, P., & Resende, M. (Eds.), *Proceedings of the Eighth International Conference on Learning and Intelligent Optimization (LION'14)*, Lecture Notes in Computer Science, pp. 36–40. Springer-Verlag.

Hutter, F., Xu, L., H.Hoos, & Leyton-Brown, K. (2014b). Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, *206*, 79–111.

Hutter, F., Xu, L., Hoos, H. H., & Leyton-Brown, K. (2014c). Algorithm runtime prediction: Methods evaluation. *Artificial Intelligence*, *206*(0), 79–111.

Janhunen, T. (2006). Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, *16*(1-2), 35–86.

Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., & Sellmann, M. (2011). Algorithm selection and scheduling. In Lee, J. (Ed.), *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'11)*, Vol. 6876 of *Lecture Notes in Computer Science*, pp. 454–469. Springer-Verlag.

Kadioglu, S., Malitsky, Y., Sellmann, M., & Tierney, K. (2010). ISAC – instance-specific algorithm configuration.. In Coelho et al. (Coelho et al., 2010), pp. 751–756.

KhudaBukhsh, A., Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2009). SATenstein: Automatically building local search SAT solvers from components.. In Boutilier (Boutilier, 2009), pp. 517–524.

Kotthoff, L. (2012). Hybrid regression-classification models for algorithm selection. In Raedt, L. D., Bessière, C., Dubois, D., Doherty, P., Frasconi, P., Heintz, F., & Lucas, P. (Eds.), *Proceedings of the Twentyfirst European Conference on Artificial Intelligence (ECAI'12)*, Vol. 242, pp. 480–485. IOS Press.

Kotthoff, L. (2013). LLAMA: leveraging learning to automatically manage algorithms. Tech. rep., Cork Constraint Computation Centre. published at arXiv.

Kotthoff, L., Gent, I. P., & Miguel, I. (2012). An evaluation of machine learning in algorithm selection for search problems. *AI Communications*, *25*(3), 257–270.

Lazaar, N., Hamadi, Y., Jabbour, S., & Sebag, M. (2012). Cooperation control in parallel SAT solving: a multi-armed bandit approach. Tech. rep., INRIA.

Li, C., Wei, W., & Li, Y. (2012). Exploiting historical relationships of clauses and variables in local search for satisfiability.. In Cimatti, & Sebastiani (Cimatti & Sebastiani, 2012), pp. 479–480.

Li, X., Garzarán, M., & Padua, D. (2007). Optimizing sorting with machine learning algorithms. In *Proceedings of the Twenty-first International Parallel and Distributed Processing Symposium (IPDPS'07)*, pp. 1–6. IEEE Computer Society Press.

Liu, G., Janhunen, T., & Niemelä, I. (2012). Answer set programming via mixed integer programming. In Brewka, G., Eiter, T., & McIlraith, S. (Eds.), *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, pp. 32–42. AAAI Press.

López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., & Birattari, M. (2011). The irace package, iterated race for automatic algorithm configuration. Tech. rep., IRIDIA, Université Libre de Bruxelles, Belgium.

Malitsky, Y., Mehta, D., & O'Sullivan, B. (2013). Evolving instance specific algorithm configuration. In Helmert, H., & Röger, G. (Eds.), *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SOCS'13)*, pp. 132–140. Proceedings of the National Conference on Artificial Intelligence (AAAI).

Malitsky, Y., Sabharwal, A., Samulowitz, H., & Sellmann, M. (2012). Parallel sat solver selection and scheduling. In Milano, M. (Ed.), *Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming (CP'12)*, Vol. 7514 of *Lecture Notes in Computer Science*, pp. 512–526. Springer-Verlag.

Malitsky, Y., Sabharwal, A., Samulowitz, H., & Sellmann, M. (2013a). Algorithm portfolios based on cost-sensitive hierarchical clustering. In Rossi, F. (Ed.), *Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence (IJCAI'13)*. IJCAI/AAAI. 608-614.

Malitsky, Y., Sabharwal, A., Samulowitz, H., & Sellmann, M. (2013b). Boosting sequential solver portfolios: Knowledge sharing and accuracy prediction.. In Pardalos, & Nicosia (Pardalos & Nicosia, 2013), pp. 153–167.

Malitsky, Y., & Sellmann, M. (2012). Instance-specific algorithm configuration as a method for non-model-based portfolio generation. In Beldiceanu, N., Jussien, N., & Pinson, E. (Eds.), *CPAIOR*, Vol. 7298 of *Lecture Notes in Computer Science*, pp. 244–259. Springer-Verlag.

Maratea, M., Pulina, L., & Ricca, F. (2012). Applying machine learning techniques to ASP solving.. In Dovier, & Santos Costa (Dovier & Santos Costa, 2012), pp. 37–48.

Maratea, M., Pulina, L., & Ricca, F. (2013). A multi-engine approach to answer-set programming. *Theory and Practice of Logic Programming*, *First View*, 1–28.

Nguyen, M., Janhunen, T., & Niemelä, I. (2013). Translating answer-set programs into bit-vector logic. In Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., & Wolf, A. (Eds.), *Proceedings of the Nineteenth International Conference on Applications of Declarative*

Programming and Knowledge Management (INAP'11) and the Twenty-fifth Workshop on Logic Programming (WLP'11), Vol. 7773 of *Lecture Notes in Computer Science*, pp. 105–116. Springer-Verlag.

Nudelman, E., Leyton-Brown, K., Hoos, H., Devkar, A., & Shoham, Y. (2004). Understanding random SAT: Beyond the clauses-to-variables ratio. In Wallace, M. (Ed.), *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'04)*, Vol. 3258 of *Lecture Notes in Computer Science*, pp. 438–452. Springer-Verlag.

O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., & O'Sullivan, B. (2008). Using case-based reasoning in an algorithm portfolio for constraint solving. In Bridge, D., Brown, K., O'Sullivan, B., & Sorensen, H. (Eds.), *Proceedings of the Nineteenth Irish Conference on Artificial Intelligence and Cognitive Science (AICS'08)*.

Pardalos, P., & Nicosia, G. (Eds.). (2013). *Proceedings of the Seventh International Conference on Learning and Intelligent Optimization (LION'13)*, Vol. 7997 of *Lecture Notes in Computer Science*. Springer-Verlag.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Petrik, M., & Zilberstein, S. (2006). Learning static parallel portfolios of algorithms. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics (ISAIM 2006)*.

Pulina, L., & Tacchella, A. (2007). A multi-engine solver for quantified boolean formulas.. In Bessiere (Bessiere, 2007), pp. 574–589.

Pulina, L., & Tacchella, A. (2009). A self-adaptive multi-engine solver for quantified Boolean formulas. *Constraints*, *14*(1), 80–116.

Rice, J. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

Roussel, O. (2011). Description of ppfolio..

Sakallah, K., & Simon, L. (Eds.). (2011). *Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'11)*, Vol. 6695 of *Lecture Notes in Computer Science*. Springer-Verlag.

Schneider, M., & Hoos, H. (2012). Quantifying homogeneity of instance sets for algorithm configuration.. In Hamadi, & Schoenauer (Hamadi & Schoenauer, 2012), pp. 190–204.

Seipp, J., Braun, M., Garimort, J., & Helmert, M. (2012). Learning portfolios of automatically tuned planners. In McCluskey, L., Williams, B., Silva, J. R., & Bonet, B. (Eds.), *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS'12)*. AAAI.

Seipp, J., Sievers, S., & Hutter, F. (2015). Automatic configuration of sequential planning portfolios. In *Proceedings of the Conference on Artificial Intelligence (AAAI'15)*. to appear.

Silverthorn, B., Lierler, Y., & Schneider, M. (2012). Surviving solver sensitivity: An ASP practitioner's guide.. In Dovier, & Santos Costa (Dovier & Santos Costa, 2012), pp. 164–175.

Simons, P., Niemelä, I., & Soininen, T. (2002). Extending and implementing the stable model semantics. *Artificial Intelligence*, *138*(1-2), 181–234.

Sinz, C. (2007). Visualizing SAT instances and runs of the DPLL algorithm. *Journal of Automated Reasoning*, *39*(2), 219–243.

Soos, M., Nohl, K., & Castelluccia, C. (2009). Extending SAT solvers to cryptographic problems. In Kullmann, O. (Ed.), *Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, Vol. 5584 of *Lecture Notes in Computer Science*, pp. 244–257. Springer-Verlag.

Streeter, M., Golovin, D., & Smith, S. (2007). Combining multiple heuristics online.. In Holte, & Howe (Holte & Howe, 2007), pp. 1197–1203.

Syrjänen, T. (2001). Lparse 1.0 user's manual..

Tamura, N., Taga, A., Kitagawa, S., & Banbara, M. (2009). Compiling finite linear CSP into SAT. *Constraints*, *14*(2), 254–272.

Thornton, C., Hutter, F., Hoos, H., & Leyton-Brown, K. (2013). Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In I.Dhillon, Koren, Y., Ghani, R., Senator, T., Bradley, P., Parekh, R., He, J., Grossman, R., & Uthurusamy, R. (Eds.), *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*, pp. 847–855. ACM.

Tompkins, D., Balint, A., & Hoos, H. (2011). Captain Jack – new variable selection heuristics in local search for SAT.. In Sakallah, & Simon (Sakallah & Simon, 2011), pp. 302–316.

van Gelder, A. (2011). Careful ranking of multiple solvers with timeouts and ties.. In Sakallah, & Simon (Sakallah & Simon, 2011), pp. 317–328.

van Gelder, A. (2012). Contrasat - a contrarian SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, *8*(1/2), 117–122.

Wei, W., & Li, C. (2009). Switching between two adaptive noise mechanism in local search for SAT.. Available at `http://home.mis.u-picardie.fr/~cli/EnglishPage.html`.

Whaley, R., Petitet, A., & Dongarra, J. (2001). Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, *27*(1-2), 3–35.

Wotzlaw, A., van der Grinten, A., Speckenmeyer, E., & Porschen, S. (2012). pfolioUZK: Solver description.. In Balint et al. (Balint et al., 2012a), p. 45. Available at `https://helda.helsinki.fi/handle/10138/34218`.

Xu, L., Hoos, H., & Leyton-Brown, K. (2007). Hierarchical hardness models for SAT.. In Bessiere (Bessiere, 2007), pp. 696–711.

Xu, L., Hoos, H., & Leyton-Brown, K. (2010). Hydra: Automatically configuring algorithms for portfolio-based selection. In Fox, M., & Poole, D. (Eds.), *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10)*, pp. 210–216. AAAI Press.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2008). SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, *32*, 565–606.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2009). SATzilla2009: An automatic algorithm portfolio for SAT. In Le Berre, D., Roussel, O., Simon, L., Manquinho, V., Argelich,

J., Li, C., Manyà, F., & Planes, J. (Eds.), *SAT 2009 competitive events booklet: preliminary version*, pp. 53–55. Available at http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2011). Detailed SATzilla Results from the Data Analysis Track of the 2011 SAT Competition. Tech. rep., University of British Columbia.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2012a). Evaluating component solver contributions to portfolio-based algorithm selectors.. In Cimatti, & Sebastiani (Cimatti & Sebastiani, 2012), pp. 228–241.

Xu, L., Hutter, F., Shen, J., Hoos, H., & Leyton-Brown, K. (2012b). SATzilla2012: Improved algorithm selection based on cost-sensitive classification models.. In Balint et al. (Balint et al., 2012a), pp. 57–58. Available at https://helda.helsinki.fi/handle/10138/34218.

Yasumoto, T. (2012). Sinn.. In Balint et al. (Balint et al., 2012a), pp. 61–61. Available at https://helda.helsinki.fi/handle/10138/34218.

Yun, X., & Epstein, S. (2012). Learning algorithm portfolios for parallel execution.. In Hamadi, & Schoenauer (Hamadi & Schoenauer, 2012), pp. 323–338.