aspeed: Solver Scheduling via Answer Set Programming∗

Holger Hoos
Department of Computer Science
University of British Columbia,
Vancouver, Canada
hoos@cs.ubc.ca

Roland Kaminski
Institute of Informatics,
University of Potsdam, Germany
kaminski@cs.uni-potsdam.de

Marius Lindauer
Institute of Informatics,
University of Potsdam, Germany
manju@cs.uni-potsdam.de

Torsten Schaub
Institute of Informatics,
University of Potsdam, Germany
tosten@cs.uni-potsdam.de

submitted 1 March 2013; revised 9 October 2013; accepted 9 December 2013

Abstract

Although Boolean Constraint Technology has made tremendous progress over the last decade, the efficacy of state-of-the-art solvers is known to vary considerably across different types of problem instances and is known to depend strongly on algorithm parameters. This problem was addressed by means of a simple, yet effective approach using handmade, uniform and unordered schedules of multiple solvers in *ppfolio*, which showed very impressive performance in the 2011 SAT Competition. Inspired by this, we take advantage of the modeling and solving capacities of Answer Set Programming (ASP) to automatically determine more refined, that is, non-uniform and ordered solver schedules from existing benchmarking data. We begin by formulating the determination of such schedules as multi-criteria optimization problems and provide corresponding ASP encodings. The resulting encodings are easily customizable for different settings and 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 solvers on hundreds to thousands of instances. Also, the fact that our approach can be customized easily enabled us to swiftly adapt it to generate parallel schedules for multi-processor machines.

KEYWORDS: Algorithm Schedules, Answer Set Programming, Portfolio-Based Solving

∗ Extended version of *aspeed: ASP based Solver Scheduling* published at ICLP'12.

1 Introduction

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 solver schedule. 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 solver 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 solvers 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 2, 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 solver (or solver configuration) a different time slice. Once such a robust schedule is found, we optimize its runtime by selecting the best solver 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 3). Once such a schedule is identified, we use a second encoding to find a time-minimal alignment of its solvers (Section 4). 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 5, we provide an empirical evaluation of the resulting system *aspeed*, and we contrast it with related approaches (Section 6). In what follows, we presuppose a basic acquaintance with ASP (see (Gebser et al. 2012) for a comprehensive introduction).

	s_1	s_2	s_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 1: Table of solver runtimes on problem instances with $\kappa = 10$; '≥ 10' indicates a timeout.

2 Solver Scheduling

In the following, we formulate the optmization problem of computing a solver schedule. To this end, we introduce robust timeout-minimal schedules for single-threaded sytems that are extended by a solver 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 solver schedules.

2.1 Sequential Scheduling

Given a set I of problem instances and a set S of solvers, we use function $t : I \times S \mapsto \mathbb{R}^+$ to represent a table of solver runtimes on instances. Also, we use an integer κ to represent a given cutoff time. For illustration, consider the runtime function in Table 1; it deals with 6 problem instances, i_1 to i_6, and 3 solvers, s_1, s_2, and s_3.

Each solver can solve three out of six instances within the cutoff time, $\kappa = 10$; timeouts are indicated by '≥ 10' in Table 1. The oracle solver, also known as virtual best solver (VBS), is obtained by assuming the best performance of each individual solver. 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 solver to choose for each instance, we could solve all of them. While we can hardly hope to practically realize an oracle solver on a single threaded system (at least in terms of CPU time), performance improvements can already be obtained by successively running each solver for a limited period of time rather than running a single solver until the cutoff is reached. For instance, by uniformly distributing time over all three solvers in our example, as done in *ppfolio*, we could solve 4 out of 6 instances, namely instance $i_1 \ldots i_4$. Furthermore, the number of solved instances can be increased further by running s_1 for 1, s_2 for 6, and s_3 for 2 seconds, which allows us to solve 5 out of 6 instances, as indicated in bold in Table 1. In what follows, we show how such a optimized non-uniform schedule can be obtained beforehand from given runtime data.

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

$$\begin{array}{c} \sigma \in \arg\max_{\sigma:S\to[0,\kappa]} |\{i \mid \exists s \in S : t(i,s) \leq \sigma(s)\}| \\ \text{such that} \qquad \sum_{s\in S}\sigma(s) \leq \kappa \end{array} \tag{1}$$

An optimal schedule σ consists of slices $\sigma(s)$ indicating the (possibly zero) time allotted to each solver $s \in S$. 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 a solver $s \in S$ that has

an equal or greater time slice $\sigma(s)$ than the time needed by the solver to solve the instance, viz. $t(i,s)$. As a side constraint, the sum of all time slices $\sigma(s)$ has to be equal or less than the cutoff time κ.

The above example corresponds to the schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 6, s_3 \mapsto 2\}$; in fact, σ constitutes one of nine timeout-optimal solver 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 solvers are also timeout-optimal. A timeout-optimal schedule consuming the entire allotted time is $\{s_1 \mapsto 0, s_2 \mapsto 8, s_3 \mapsto 2\}$.

In practice, however, the criterion in (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[1] as $\Sigma_{s\in S,\sigma(s)\neq 0}\,\sigma(s)^n$. 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) solvers 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 solver a similar time slice, while maximizing it prefers schedules with large runtimes for few solvers. In more formal terms, for a given set S of solvers, using an L^n-norm we would like to determine schedules satisfying the constraint

$$\sigma \in \arg\min_{\sigma:S\to[0,\kappa]} \Sigma_{s\in S,\sigma(s)\neq 0}\,\sigma(s)^n, \tag{2}$$

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

For instance, our example schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 6, s_3 \mapsto 2\}$ 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 $\{s_1 \mapsto 3, s_2 \mapsto 3, s_3 \mapsto 3\}$ and 1, 9, and 81 for a singular schedule $\{s_3 \mapsto 9\}$, 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, 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 solver schedules.

Once we have identified an optimal schedule w.r.t. criteria (1) and (2), it is interesting to determine which solver alignment yields the best performance as regards time. More formally, we define an *alignment* of a set S of solvers as a bijective function $\pi : \{1,\ldots,|S|\} \to S$. Consider the above schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 6, s_3 \mapsto 2\}$. The alignment $\pi = \{1 \mapsto s_1, 2 \mapsto s_3, 3 \mapsto s_2\}$ induces the execution sequence (s_1, s_3, s_2) of σ. This sequence takes 29 seconds for all six benchmarks in Table 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 solver in the alignment, viz. s_2. Hence the total time amounts to the time allotted by σ to s_1 and s_3, viz. $\sigma(s_1)$ and $\sigma(s_3)$, plus the effective time of s_2, viz. $t(i_3, s_2)$.

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

$$\tau_{\sigma,\pi}(i) = \begin{cases} \left(\sum_{j=1}^{\min(P_{\sigma,\pi})-1} \sigma(\pi(j))\right) + t(i, \pi(\min(P_{\sigma,\pi}))) & \text{if } P_{\sigma,\pi} \neq \emptyset, \\ \kappa & \text{otherwise} \end{cases} \tag{3}$$

[1] 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.

where $P_{\sigma,\pi} = \{l \in \{1,\ldots,|S|\} \mid t(i,\pi(l)) \leq \sigma(\pi(l))\}$ are the positions of solvers 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 κ. 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 solvers in S, we then define the optimal aligment of schedule σ:

$$\pi \in \arg\min_{\pi:\{1,\ldots,|S|\}\to S} \textstyle\sum_{i\in I} \tau_{\sigma,\pi}(i) \tag{4}$$

For our timeout-optimal schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 6, s_3 \mapsto 2\}$ w.r.t. criteria (1) and (2), we obtain two optimal execution alignments, namely (s_3, s_1, s_2) and (s_1, s_3, s_2), both of which result in a solving time of 29 seconds for the benchmarks of Table 1.

2.2 Parallel Scheduling

The increasing availability of multi-core processors makes it interesting to extend our approach for distributing a schedule's solvers over multiple processing units. For simplicity, we take a coarse approach in binding solvers 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 solvers from S. More formally, we define a *distribution* of a set S of solvers as the function $\eta : U \to 2^S$ such that $\bigcap_{u\in U} \eta(u) = \emptyset$. With it, we can determine timeout-optimal solver schedules for several cores simply by strengthening the condition in (1) to the effect that all solvers associated with the same unit must respect the cutoff time. This leads us to the following extension of (1):

$$\begin{aligned} &\sigma \in \arg\max_{\sigma:S\to[0,\kappa]} |\{i \mid \exists s \in S : t(i,s) \leq \sigma(s)\}| \\ &\quad \text{such that} \qquad \textstyle\sum_{s\in\eta(u)} \sigma(s) \leq \kappa \text{ for each } u \in U \end{aligned} \tag{5}$$

For illustration, let us reconsider Table 1 along with schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 8, s_3 \mapsto 2\}$. Assume that we have two cores, 1 and 2, along with the distribution $\eta = \{1 \mapsto \{s_2\}, 2 \mapsto \{s_1, s_3\}\}$. This distributed schedule is an optimal solution to the optimization problem in (5) w.r.t. the benchmarks in Table 1 because it solves all benchmarks within a cutoff time of $\kappa = 8$.

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 (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 solvers 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 (3) as follows:

$$\tau_{\sigma,\pi_u}(i) = \begin{cases} \left(\sum_{j=1}^{\min(P_{\sigma,\pi})-1} \sigma(\pi_u(j))\right) + t(i,\pi_u(\min(P_{\sigma,\pi}))) & \text{if } P_{\sigma,\pi} \neq \emptyset, \\ \kappa & \text{otherwise} \end{cases} \tag{6}$$

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 solvers in S and a set U of (processing) units, we then define an optimal global alignment:

$$(\pi_u)_{u\in U} \in \arg\min_{(\pi_u:\{1,\ldots,|\eta(u)|\}\to\eta(u))_{u\in U}} \textstyle\sum_{i\in I} \min_{u\in U} \tau_{\sigma,\pi_u}(i) \tag{7}$$

For illustration, reconsider the above schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 8, s_3 \mapsto 2\}$ and distribution

$\eta = \{1 \mapsto \{s_2\}, 2 \mapsto \{s_1, s_3\}\}$, and suppose we chose the local alignments $\pi_1 = \{1 \mapsto s_2\}$ and $\pi_2 = \{1 \mapsto s_1, 2 \mapsto s_3\}$. This global alignment solves all six benchmark instances of Table 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 (5), (6), and (7) correspond to their sequential counterparts in (1), (3), and (4) whenever we are faced with a single processing unit.

3 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 $t : I \times S \mapsto \mathbb{R}$ is represented as facts of form `time(i,s,t)`, where `i` $\in I$, `s` $\in S$, and `t` is the runtime $\texttt{t}(\texttt{i}, \texttt{s})$, converted to a natural number with limited precision. The cutoff is expressed via Predicate `kappa/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 1 as shown in Listing 1 below.

Listing 1: Facts

```
kappa(10).
units(2).

time(i1, s1,  1).  time(i1, s2, 11).  time(i1, s3,  3).
time(i2, s1,  5).  time(i2, s2, 11).  time(i2, s3,  2).
time(i3, s1,  8).  time(i3, s2,  1).  time(i3, s3, 11).
time(i4, s1, 11).  time(i4, s2, 11).  time(i4, s3,  2).
time(i5, s1, 11).  time(i5, s2,  6).  time(i5, s3, 11).
time(i6, s1, 11).  time(i6, s2,  8).  time(i6, s3, 11).
```

The encoding in Listing 3 along with all following ones are given in the input language of *gringo* (Gebser et al.). The first three lines of Listing 3 provide auxiliary data. The set S of solvers is given by Predicate `solver/1`. Similarly, the runtimes for each solver are expressed by `time/2` and each processing unit by `unit/1`. In addition, the ordering of instances by time per solver is precomputed; it is expressed via `order/3`.

Listing 2: `I` is solved immediatly before `J` by solver `S`

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

The above results in facts `order(I,J,S)` 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 3 is now to guess for each solver 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).

Listing 3: ASP encoding for Timeout-Minimal (Parallel) Scheduling

```
1  solver(S)  :- time(_,S,_).
2  time(S,T)  :- time(_,S,T).
3  unit(1..N) :- units(N).

5  {slice(U,S,T): time(S,T): T <= K: unit(U)} 1 :- solver(S),kappa(K).
6   :- not [ slice(U,S,T) = T ] K, kappa(K), unit(U).

8  slice(S,T) :- slice(_,S,T).

10 solved(I,S) :- slice(S,T), time(I,S,T).
11 solved(I,S) :- solved(J,S), order(I,J,S).
12 solved(I)   :- solved(I,_).

14 #maximize { solved(I) @ 2 }.
15 #minimize [ slice(S,T) = T*T @ 1 ].
```

In more detail, a schedule is represented by atoms `slice(U,S,T)` allotting a time slice `T` to solver `S` on unit `U`. In Line 5, at most one time slice is chosen for each solver, 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 solver. 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 (5), and it reduces to the one in (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 solver `S`; using it, along with the sorting of instances by solver performance in `order/3`, we collect in Line 11 all 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(S,T), time(I,S,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 (5) (or (1), respectively). This primary objective is assigned a higher priority than the L^2-norm from (2) (priority 2 *vs* 1).

4 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 a solver alignment per processing unit while minimizing the overall runtime according to Criterion (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 `kappa/1`, `units/1`, and `time/3`) by facts over `slice/3` providing the time slices of a timeout-minimal schedule (per solver and processing unit). In the case of our example from Section 2.2, we extend the facts of Listing 1 with the following obtained timeout-minimal schedule to create the problem instance:

Listing 4: Schedule Facts

```
slice(1,s2,8). slice(2,s1,1). slice(2,s3,2).
```

The idea of the encoding in Listing 5 is to guess a permutation of solvers 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.

The block in Line 1 to 10 gathers static knowledge about the problem instance, that is, solvers per processing unit (`solver/2`), instances appearing in the problem description (`instance/1`), available processing units (`unit/1`), number of solvers per unit (`solvers/2`), instances solved by a solver 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 3), the solved instances (`solved/3`) can be efficiently expressed as done in Line 5 of Listing 5, because `slice/3` are facts here. In view of Equation (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 solver `S` on unit `U` a time `T`. In Line 7, we assign the time needed to solve the instance if it is within the solver's time slice. In Line 8, we assign the solver's time slice if the instance could not be solved, but at least one other solver could solve it on processing unit `U`. In Line 9, we assign the entire cutoff to dummy solver `d` (we assume that there is no other solver called `d`) if the instance could not be solved on the processing unit at all; this is to implement the else case in (6) and (3).

The actual encoding starts in Line 12 and 13 by guessing a permutation of solvers. Here, the two head aggregates ensure that for every solver (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 solver 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 (6) and (3). Next, we determine the smallest sum of times by iteratively determining the minimum. An atom `min(U,I,S,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 solver `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

Listing 5: ASP encoding for Time-Minimal (Parallel) Scheduling

```
1  solver(U,S)      :- slice(U,S,_).
2  instance(I)      :- time(I,_,_).
3  unit(1..N)       :- units(N).
4  solvers(U,N)     :- unit(U), N := {solver(U,_)}.
5  solved(U,S,I)    :- time(I,S,T), slice(U,S,TS), T <= TS.
6  solved(U,I)      :- solved(U,_,I).
7  capped(U,I,S,T)  :- time(I,S,T), solved(U,S,I).
8  capped(U,I,S,T)  :- slice(U,S,T), solved(U,I), not solved(U,S,I).
9  capped(U,I,d,K)  :- unit(U), kappa(K), instance(I), not solved(U,I).
10 capped(I,S,T)    :- capped(_,I,S,T).

12 1 { order(U,S,X) : solver(U,S) } 1 :- solvers(U,N), X = 1..N.
13 1 { order(U,S,X) : solvers(U,N) : X = 1..N } 1 :- solver(U,S).

15 solvedAt(U,I,X+1) :- solved(U,S,I), order(U,S,X).
16 solvedAt(U,I,X+1) :- solvedAt(U,I,X), solvers(U,N), X <= N.

18 mark(U,I,d,K) :- capped(U,I,d,K).
19 mark(U,I,S,T) :- capped(U,I,S,T), order(U,S,X), not solvedAt(U,I,X).
20 min(1,I,S,T)  :- mark(1,I,S,T).

22 less(U,I) :- unit(U), unit(U+1), instance(I),
23   [min(U,I,S1,T1): capped(I,S1,T1) = T1, mark(U+1,I,S2,T2) = -T2] 0.

25 min(U+1,I,S,T) :- min(U,I,S,T), less(U,I).
26 min(U,I,S,T)   :- mark(U,I,S,T), not less(U-1,I).

28 #minimize [min(U,_,_,T): not unit(U+1) = T].
```

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 (7) and (4).

5 Experiments

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 et al. 2011) of the potassco group (Gebser et al. 2011),

	Random	*Crafted*	*Application*	*ASP-Set*
Cutoff (sec.)	5000	5000	5000	900
#Instances	600	300	300	2589
#Solvers	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
#Solvers	37	2	5	11
Source	(3)	(4)	(5)	(6)

Table 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).

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. [2]

5.1 Experimental setup

Our experiments are based on a set of runtime data obtained by running several solvers (or solver configurations) on a set of benchmark instances (similar to Table 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 2. The sets *Random*, *Crafted* and *Application* contain the authentic runtimes taken from the 2011 SAT Competition[3] 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. 2012).

Also, we evaluated our approach on an ASP instance set (*ASP-Set*) based on different configurations of the highly parametric ASP solver *clasp* (Gebser et al. 2012), 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*,[4] 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,

[2] `http://www.cs.uni-potsdam.de/aspeed`
[3] `http://www.cril.univ-artois.fr/SAT11`
[4] `http://asparagus.cs.uni-potsdam.de`

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 et al. (2010), the *QBF-Set* by Pulina and Tacchella (2009), and *MaxSAT-Set* by Malitsky et al. (2013), respectively.

The performance of *aspeed* was determined from the schedules computed for Encodings 3 and 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 solvers in the schedule.

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

- *single best*: the best solver in the respective portfolio,
- *uniform*: a uniform distribution of the time slices over all solvers in the portfolio,
- *ppfolio-like*: an approach inspired by *ppfolio*, where the best three complementary solvers are selected with an uniform distribution of time slices in the sequential case,
- *satzilla* (Xu et al. 2012) 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)[5].

The performance of *satzilla* for *Random*, *Crafted* and *Application* was extracted from results reported in the literature (Xu et al. 2012), 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.

5.2 Schedule Computation

Table 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

[5] The performance of the *oracle* is the minimal runtime of each instance given a portfolio of solvers and corresponds to a portfolio-based solver with a perfect selection of the best solver 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 3: Runtimes of *clasp* in CPU seconds to calculate an optimal schedule for one and eight cores.

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 solvers 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 solvers on the individual cores than in the sequential case. This leads to smaller permutations of solvers 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.[6]

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 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*.

[6] Note that in our experiments, the performance of *unclasp* (Andres et al. 2012), which optimizes based on unsatisfiable cores, did not exceed the perfomance of *clasp* in computing solver 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: Comparison of different approaches w.r.t. #timeouts / #instances. The performance of the best performing system is in boldface.

5.3 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 (1)). The number of timeouts for a fixed time budget assesses the robustness of a solver and is in many applications and competitions the primary evaluation criterion.

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, $9/10$th of the data is used as a training set for the computation of the schedule and the remaining $1/10$th serves as a test set to evaluate the performance of the solver 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 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* solver. 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 solver 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.

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

5.4 Evaluation of Time-Minimal Alignment

After choosing the time slices for each solver, it is necessary to compute an appropriate solver 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 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.

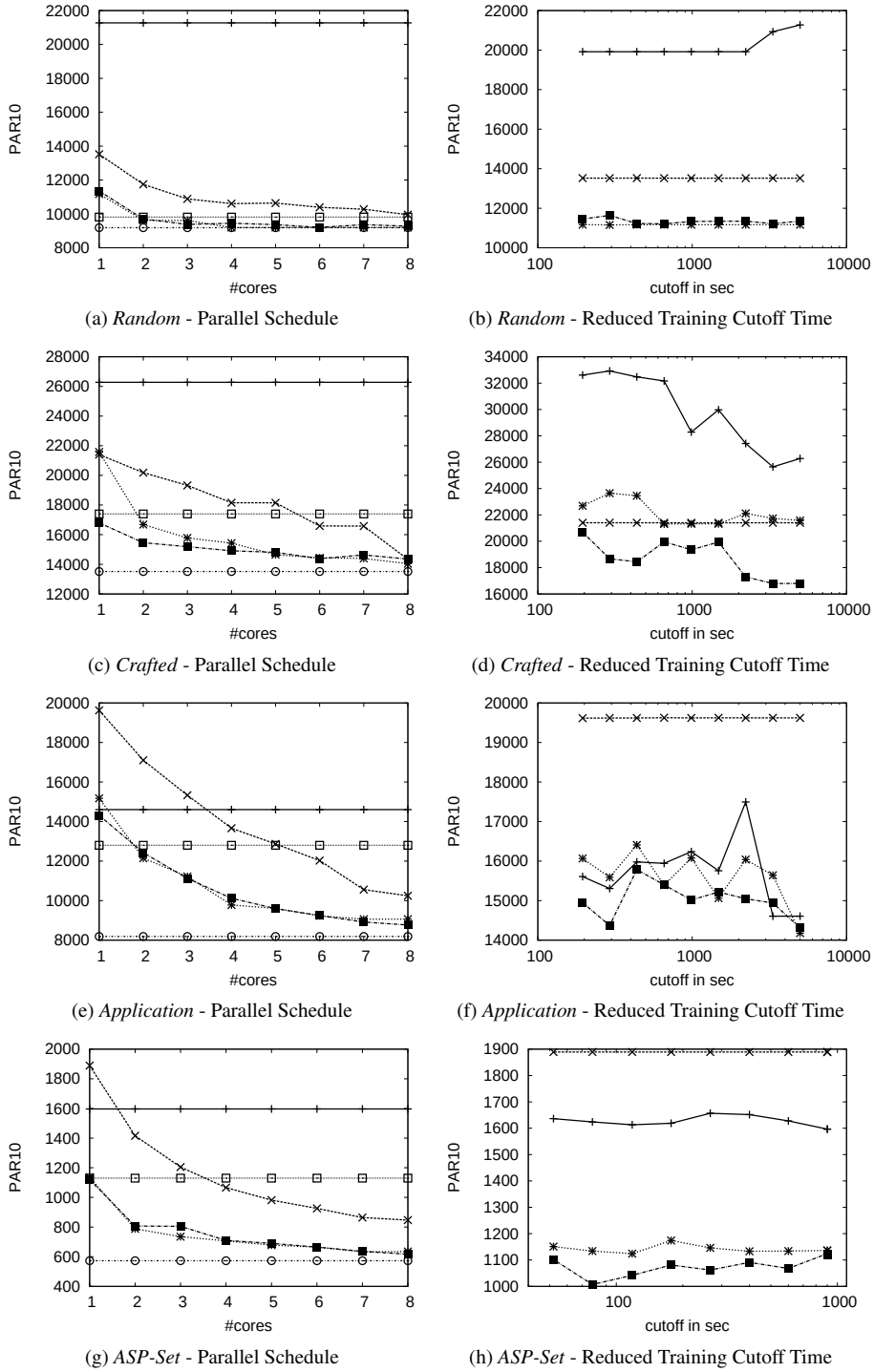
Also, we contrast the optimal alignment with two easily computable heuristic alignments to avoid the search for an optimal alignment. The alignment heuristic *heu-Opt* sorts solvers beginning with the solver with the minimal number of timeouts (most robust solver), while *heu-Min* begins with the solver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

As expected, the best performance is obtained by using optimal alignments within *aspeed* (Table 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) (data not shown). The penalized runtimes (PAR10) are presented in Figure 1 (a),(c) and (e) at $\#cores = 1$.

5.5 Parallel Schedules

As we have seen in Section 3, 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



(a) *Random* - Parallel Schedule (b) *Random* - Reduced Training Cutoff Time

(c) *Crafted* - Parallel Schedule (d) *Crafted* - Reduced Training Cutoff Time

(e) *Application* - Parallel Schedule (f) *Application* - Reduced Training Cutoff Time

(g) *ASP-Set* - Parallel Schedule (h) *ASP-Set* - Reduced Training Cutoff Time

Fig. 1: Parallel Schedules (left) and reduced cutoff time (right), *single best* (+), *uniform* (×), *ppfolio-like* approach (∗), *aspeed* (■), *selection* (□), *oracle* (◯).

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

presented in Figure 1 (a), (c), (e) and (g). 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 runtime)[7] 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* solver ($+$) and *selection* ($\square$, *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 solver on the first core with uniform time slices and one component solvers on each other core. This more constrained schedule is also computed with the ASP encodings presented in Section 3 by adding three more constraints.

As stated previously, the sequential version of *aspeed* ($\blacksquare$) performed worse than *satzilla* ($\square$) 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 of *satzilla* (Xu et al. 2012) and *claspfolio* (Gebser et al. 2011), this represents a substantial performance improvement.

Table 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 solvers, and therefore it is trivial to perform as well as the *oracle* with 4 or more cores, and (ii) we saw in Figure 1 that the curves flatten beginning with four cores, which is an effect of the complementarity of the solvers in the portfolio. The performance of *aspeed-SP*, i.e., sequential *aspeed*, is already nearly as good as the *oracle* on *MaxSAT-Set* and *aspeed-4P* was only able

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

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

Table 7: PAR10 of *single best* and *aspeed*, trained on 2009 SAT Competition and evaluated on 2011 SAT Competition.

to improve the performance slightly. However, *aspeed-4P* was able to decrease the number of timeouts from 603 to 204 on the *3S-Set*.

5.6 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 solver 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 1 (b), (d) and (f) 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 ($\times$) 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 1b) and *ASP-Set* (Figure 1h). On the other hand, the selection of the *single best* solver ($+$) got worse with an increased cutoff time on the training data of *Random*. On the *Crafted* set (Figure 1d), 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 1f). 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 7. On the entire instance set, *aspeed*

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

Table 8: PAR10 of *3S* and *aspeed*, trained on the training data of *3S* and evaluated on 2011 SAT Competition.

had a PAR10 of 21 196, in contrast to the *single best* solver with 32 457 (a factor 1.53 higher). Also, *aspeed* outperformed the *single best* solver on *Random* and *Crafted*, and it performed just as well as the *single best* solver 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*.

5.7 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 solver schedule with algorithm selection based on instance features (see Section 6). 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.

Table 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 solvers on these types of instances. Furthermore, *aspeed* with four cores, *aspeed-4P*, performed better than *3S* on all three instance sets.

6 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 et al. 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 and Milano ; Gebruers et al. 2004); and work on various machine learning techniques for selecting algorithms

for finding maximum probable explanations in Bayes nets in real time (Guo and 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*, i.e., 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 and 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 et al. 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 *Stone Soup* (Helmert et al. 2011; Seipp et al. 2012) which statically schedules planners. In contrast to *aspeed*, *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 *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.

7 Conclusion

In this work, we demonstrated how ASP formulations and a powerful ASP solver (*clasp*) can be used to compute sequential and parallel solver 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*, respectively), 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 solver, 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 (e.g., competitions) or where single good (or even dominant) solvers or solver configurations are unknown (e.g., 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 a solver 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 solver). 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 solvers or configurations, and it would be interesting to investigate this dependence quantitatively, possibly based on recently proposed formal definitions of instance set homogeneity (Schneider and Hoos 2012). Alternatively, if a dominant solver configuration is expected to exist but is unknown, such a configuration could be found using an algorithm configurator, for instance *ParamILS* (Hutter et al. 2007; Hutter et al. 2009), *GGA* (Ansótegui et al. 2009), *F-Race* (López-Ibáñez et al. 2011) or *SMAC* (Hutter et al. 2011). Furthermore, automatic methods, like *hydra* (Xu et al. 2010) and *isac* (Kadioglu et al. 2010), construct automatically complementary portfolios of solver configurations with the help of algorithm configurators which could be also combined with *aspeed* to further increase its performance.

Acknowledgments This work was partially funded by the German Science Foundation (DFG) under grant SCHA 550/8-3.

References

ANDRES, B., KAUFMANN, B., MATHEIS, O., AND SCHAUB, T. 2012. Unsatisfiability-based optimization in clasp. In *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12)*, A. Dovier and V. Santos Costa, Eds. Vol. 17. Leibniz International Proceedings in Informatics (LIPIcs), 212–221.

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

BARAL, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

BIERE, A., HEULE, M., VAN MAAREN, H., AND WALSH, T., Eds. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

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., AND VELTRI, P. 2011. The third answer set programming competition: Preliminary report of the system competition track. See Delgrande and Faber (2011), 388–403.

COELHO, H., STUDER, R., AND WOOLDRIDGE, M., Eds. 2010. *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10)*. IOS Press.

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

GAGLIOLO, M. AND SCHMIDHUBER, J. 2006. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence 47,* 3-4, 295–328.

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

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND 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., AND THIELE, S. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND 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., AND ZILLER, S. 2011. A portfolio solver for answer set programming: Preliminary report. See Delgrande and Faber (2011), 352–357.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187-188*, 52–89.

GENT, I., JEFFERSON, C., KOTTHOFF, L., MIGUEL, I., MOORE, N., NIGHTINGALE, P., AND PETRIE, K. 2010. Learning when to use lazy learning in constraint solving. See Coelho et al. (2010), 873–878.

GOMES, C. AND SELMAN, B. 2001. Algorithm portfolios. *Artificial Intelligence 126,* 1-2, 43–62.

GUERRI, A. AND MILANO, M. Learning techniques for automatic algorithm portfolio selection. 475–479.

GUO, H. AND 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*. Springer, 307–318.

HAMADI, Y. AND SCHOENAUER, M., Eds. 2012. *Proceedings of the Sixth International Conference Learning and Intelligent Optimization (LION'12)*. Lecture Notes in Computer Science, vol. 7219. Springer-Verlag.

HELMERT, M., RÖGER, G., AND KARPAS, E. 2011. Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS 2011 Workshop on Planning and Learning*. 28–35.

HOOS, H., KAUFMANN, B., SCHAUB, T., AND SCHNEIDER, M. 2013. Robust benchmark set selection for boolean constraint solvers. In *Proceedings of the Seventh International Conference on Learning and Intelligent Optimization (LION'13)*, P. Pardalos and G. Nicosia, Eds. Lecture Notes in Computer Science. Springer-Verlag, 138–152.

HUBERMAN, B., LUKOSE, R., AND HOGG, T. 1997. An economic approach to hard computational problems. *Science 275*, 51–54.

HUTTER, F., HOOS, H., AND LEYTON-BROWN, K. 2011. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization (LION'11)*. Lecture Notes in Computer Science, vol. 6683. Springer-Verlag, 507–523.

HUTTER, F., HOOS, H., LEYTON-BROWN, K., AND STÜTZLE, T. 2009. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research 36*, 267–306.

HUTTER, F., HOOS, H., AND STÜTZLE, T. 2007. Automatic algorithm configuration based on local search. 2007. *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07)*. AAAI Press., 1152–1157.

KADIOGLU, S., MALITSKY, Y., SABHARWAL, A., SAMULOWITZ, H., AND SELLMANN, M. 2011. Algorithm selection and scheduling. In *Proceedings of the Seventeenth International Conference on Principles*

and Practice of Constraint Programming (CP'11), J. Lee, Ed. Lecture Notes in Computer Science, vol. 6876. Springer-Verlag, 454–469.

KADIOGLU, S., MALITSKY, Y., SELLMANN, M., AND TIERNEY, K. 2010. ISAC – instance-specific algorithm configuration. See Coelho et al. (2010), 751–756.

LÓPEZ-IBÁÑEZ, M., DUBOIS-LACOSTE, J., STÜTZLE, T., AND BIRATTARI, M. 2011. The irace package, iterated race for automatic algorithm configuration. Tech. rep., IRIDIA, Université Libre de Bruxelles, Belgium.

MALITSKY, Y., MEHTA, D., AND OSULLIVAN, B. 2013. Evolving instance specific algorithm configuration. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SOCS'13)*, H. Helmert and G. Röger, Eds. Proceedings of the National Conference on Artificial Intelligence (AAAI), 132–140.

MALITSKY, Y., SABHARWAL, A., SAMULOWITZ, H., AND SELLMANN, M. 2012. Parallel sat solver selection and scheduling. In *Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming (CP'12)*, M. Milano, Ed. Lecture Notes in Computer Science, vol. 7514. Springer-Verlag, 512–526.

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

PULINA, L. AND 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.

SCHNEIDER, M. AND HOOS, H. 2012. Quantifying homogeneity of instance sets for algorithm configuration. See Hamadi and Schoenauer (2012), 190–204.

SEIPP, J., BRAUN, M., GARIMORT, J., AND HELMERT, M. 2012. Learning portfolios of automatically tuned planners. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS'12)*, L. McCluskey, B. Williams, J. R. Silva, and B. Bonet, Eds. AAAI, 368–372.

STREETER, M., GOLOVIN, D., AND SMITH, S. 2007. Combining multiple heuristics online. 2007. *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07)*. AAAI Press., 1197–1203.

TAMURA, N., TAGA, A., KITAGAWA, S., AND BANBARA, M. 2009. Compiling finite linear CSP into SAT. *Constraints 14,* 2, 254–272.

XU, L., HOOS, H., AND LEYTON-BROWN, K. 2007. Hierarchical hardness models for SAT. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07)*, C. Bessiere, Ed. Lecture Notes in Computer Science, vol. 4741. Springer-Verlag, 696–711.

XU, L., HOOS, H., AND LEYTON-BROWN, K. 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10)*, M. Fox and D. Poole, Eds. AAAI Press, 210–216.

XU, L., HUTTER, F., HOOS, H., AND 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., AND LEYTON-BROWN, K. 2012. Evaluating component solver contributions to portfolio-based algorithm selectors. In *Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, A. Cimatti and R. Sebastiani, Eds. Lecture Notes in Computer Science, vol. 7317. Springer-Verlag, 228–241.

YUN, X. AND EPSTEIN, S. 2012. Learning algorithm portfolios for parallel execution. See Hamadi and Schoenauer (2012), 323–338.