aspeed: ASP-based Solver Scheduling

Holger Hoos[1], Roland Kaminski[2], Torsten Schaub[3], and Marius Schneider[4]

1 **University of British Columbia, Canada**
`hoos@cs.ubc.ca`
2 **University of Potsdam, Germany**
`kaminski@cs.uni-potsdam.de`
3 **University of Potsdam, Germany**
`torsten@cs.uni-potsdam.de`
4 **University of Potsdam, Germany**
`manju@cs.uni-potsdam.de`

Abstract

Although Boolean Constraint Technology has made tremendous progress over the last decade, it suffers from a great sensitivity to search configuration. This problem was impressively counterbalanced at the 2011 SAT Competition by the rather simple approach of *ppfolio* relying on a handmade, uniform and unordered solver schedule. Inspired by this, we take 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. 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, its high customizability made it easy to generate even parallel schedules for multi-core machines.

1998 ACM Subject Classification I.2.8 Problem Solving, Control Methods, and Search

Keywords and phrases Algorithm Schedule, Portfolio-based Solving, Answer Set Programming

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

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; [3]), it meanwhile also led to significant boosts in neighboring areas, like Answer Set Programming (ASP; [2]), Pseudo-Boolean Solving [3, Chapter 22], and even (multi-valued) Constraint Solving [21]. However, there is yet 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 laid bare once more at the 2011 SAT competition, where 16 prizes were won by the portfolio-based solver *ppfolio* [17]. 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 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 more processing units are available, one solver is in turn started on each unit; though multiple ones may end up on the last unit.

Inspired by this plain, yet successful system, we provide a more elaborate, yet still simple approach that takes advantage of the modeling and solving capacities of ASP to automatically determine more

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

	s_1	s_2	s_3	$oracle$
i_1	**1**	(11)	3	1
i_2	5	(11)	**2**	2
i_3	8	**1**	(11)	1
i_4	(11)	(11)	**2**	2
i_5	(11)	**6**	(11)	6
i_6	(11)	8	(11)	8
timeouts	3	3	3	0

Table 1 Table of solver runtimes on problem instances with $\kappa = 10$.

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-core 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. Unlike both, our approach does not rely on any domain-specific features, which makes it easily adaptable to other problems.

The remainder of this article is structured as follows. In Section 2, we formulate the determination of optimum schedules as multi-criteria optimization problems. 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 next extend this approach to parallel settings in which multiple processing units are available. With these specifications 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 the encoding to find a time-minimal alignment of its solvers (Section 4). Both ASP encodings reflect interesting features needed for dealing with large sets of runtime data. Finally, in Section 5, we provide a 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 [2] for a comprehensive introduction).

2 Solver Scheduling

Sequential Scheduling. Given a set I of problem instances and a set S of solver configurations, 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, $\kappa = 10$. A timeout is represented in Table 1 by 11, that is, an increased cutoff time. The oracle solver, also called virtually best solver, is obtained by assuming the best performance of each individual solver. As we see in the rightmost column, the oracle would allow for solving 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. Unlike this, we can already obtain an improvement by successively running each solver within a limited time slice rather than running one solver until cutoff. For instance, running s_1 for 1, s_2 for 6, and s_3 for 2 seconds allows us to solve 5 out of 6 instances, as indicated in bold in Table 1. In what follows, we show how such a 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 an unordered tuple σ, represented as a function $\sigma : S \to [0, \kappa]$, satisfying the following condition:

$$\begin{aligned} \sigma \in \arg\max_{\sigma:S\to[0,\kappa]} & \left|\{i \mid t(i,s) \leq \sigma(s), (i,s) \in I \times S\}\right| \\ \text{such that} \quad & \textstyle\sum_{s\in S}\sigma(s) \leq \kappa \end{aligned} \tag{1}$$

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 obtained timeouts.

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 among 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 yields many heterogeneous solutions among which we would like to make an educated choice. To this end, we take advantage of L-norms for regulating the selection. In our case, an L^n-norm on schedules is defined as[1] $\Sigma_{s\in S,\sigma(s)\neq 0}\,\sigma(s)^n$. Depending upon 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/many solvers as possible and L^1-norms aim at minimizing/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, an L^n-norm gives rise to objective functions of the following form. For a set of schedules of a set S of solvers, we define:

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

An analogous function is obtained for maximization with $\arg\max$.

For instance, our exemplary schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 6, s_3 \mapsto 2\}$ has the L^i-norms 3, 9, and 41 for $i = 0..2$. For a complement, we get 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 we empirically discovered no clear edge of the latter, we favor 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 most homogeneous distribution of time slices, similar to *ppfolio*. In fact, our exemplary schedule has the smallest L^2-norm among all nine timeout-optimal solver schedules.

Once we have identified a most robust schedule wrt criteria (1) and (2), it is interesting to know which solver alignment yields the best performance as regards time. More formally, we define an *alignment* of a set S of solvers as the 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 solves all six benchmarks in Table 1 in 29 seconds; in detail, it takes $1, 1+2, 1+2+1, 1+2, 1+2+6, 1+2+6$ seconds for benchmark i_k for $k = 1..6$. Note that benchmark i_3 is successfully solved by the third solver in the alignment, viz. s_2. Hence the total time amounts to the allotted time by σ to s_1 and s_3, viz. $\sigma(s_1)$ and $\sigma(s_3)$, plus the effective time of s_2, viz. $t(i_3, s_2)$. Because the timeout-minimal time slices are given, we do not distinguish whether an alignment solves a benchmark after the total time of the schedule or not. For instance, our exemplary alignment π takes 9 seconds on both i_5 and i_6, although it only solves the former but not the latter.

---

[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.

This can be made precise as follows. Given a schedule σ and an alignment π of a set S of solvers, and an instance $i \in I$, we define:

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

where $P = \{l \in \{1, \ldots, |S|\} \mid t(i, \pi(l)) \leq \sigma(\pi(l))\}$. While $\min P$ gives the position of the first solver solving instance i in a schedule σ aligned by π, $\tau_{\sigma,\pi}(i)$ gives the total time to solve instance i by 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 exemplary schedule σ and its alignment π, we get for i_3: $\min P = 3$ and $\tau_{\sigma,\pi}(i_3) = 1 + 2 + 1 = 4$.

For a schedule σ of solvers in S, we then define:

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

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

Parallel Scheduling. The increasing availability of multi-core processors makes it interesting to extend our approach for distributing a schedule's solvers on different 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 ensemble of concepts 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):

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

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 solves all benchmarks in Table 1 with a cutoff of $\kappa = 8$. Hence, it is an optimal solution to the optimization problem in (5).

We keep the definitions of a schedule's L^n-norm as a global constraint.

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 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 and an instance $i \in I$, we extend the definitions in (3) as follows:

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

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

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:

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

For illustration, reconsider the above distribution and suppose we chose the local aligments $\pi_1 = \{s_2 \mapsto 1\}$ and $\pi_2 = \{s_1 \mapsto 1, s_3 \mapsto 2\}$. This global alignment solves all six benchmark instances in 22 seconds. In more detail, it takes $1_2, 1+2_2, 1_1, 1+2_2, 6_1, 8_1$ seconds for benchmark 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 $t(\texttt{i},\texttt{s})$ converted to a natural number with a 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 with 2 cores. Given this, we can represent the contents of Table 1 as follows.

```
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 1 along with all following ones are given in the input language of *gringo*, documented in [7]. The first three lines of Listing 1 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`. In addition, the ordering `order/3` of instances by time per solver is precomputed.

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

The above results in facts `order(I,K,S)` capturing that instance `I` is solved immediately before instance `K` by solver `S`. 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).[2]

The idea is now to guess for each solver a time slice and a processing unit. With the resulting schedule, all solvable instances can be identified. And finally all schedules solving most instances are selected.

Listing 1 ASP encoding for Timeout-Minimal (Parallel) Scheduling

```
1 solver(S)  :- time(_,S,_).
2 time(S,T)  :- time(_,S,T).
```

[2] To be precise, we use *gringo*'s embedded scripting language *lua* for sorting.

```
3  unit(1..N) :- units(N).

5  {slice(U,S,T): time(S,T): T <= K: unit(U)} 1 :- solver(S),kappa(K).
6  slice(S,T) :- slice(_,S,T).

8   :- not [ slice(U,S,T) = T ] K, kappa(K), unit(U).

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 ].
```

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 trivial condition that it is equal or less the cutoff time. At the same time, a processing unit is uniquely assigned to the selected solver. The following line projects out the processing unit because it is irrelevant when determining solved instances (in Line 10). The integrity constraint in Line 8 ensures that the sum over all selected time slices on each core 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. In lines 10 to 12, all instances solved by the selected time slices are gathered via predicate `solved/1`. Given that we collect in Line 6 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`. Given this and 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 this 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. Unlike this, our ordering-based encoding is linear because only successive instances are considered. Finally, the number of solved instances is maximized in Line 14, following the recipe in (5) (or (1), respectively). This major objective gets a higher priority, viz. `2`, than the L^2-norm from (2) having priority `1`.

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

In the previous section, we have determined a timeout-minimal 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 problem instance of 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). To take on our example from Section 2, we use the obtained timeout-minimal schedule to create the following problem instance:

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

```
time(i1, s1,  1).  time(i1, s2, 11).  time(i1, s3,  3).
...
slice(1,s2,8). slice(2,s1,1). slice(2,s3,2).
```

The idea of the encoding in Listing 2 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 fail to solve common problems with thousands of benchmark instances. To do this, we make use of `#sum` aggregates with negative weights to find the fastest processing unit without representing any sum of times explicitly.

Listing 2 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].
```

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`). 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 the processing unit. In Line 9, we assign the whole 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).

	Random		*Crafted*		*Application*		*3s-Set*		*ASP-Set*	
Single Best	254	(42.3%)	155	(51.6%)	85	(28.3%)	1881	(34.4%)	28	(8.9%)
Uniform	155	(25.8%)	123	(41.5%)	116	(38.6%)	1001	(18.3%)	29	(9.2%)
ppfolio-like	127	(21.1%)	126	(42.0%)	82	(27.3%)	645	(11.8%)	17	(5.4%)
satzilla	115	(19.2%)	101	(34.0%)	74	(24.7%)	−−	(−%)	−−	(−%)
aspeed (seq)	131	(21.8%)	98	(32.6%)	83	(27.6%)	536	(9.8%)	18	(5.7%)
aspeed (par 8)	109	(18.2%)	85	(28.3%)	51	(17.0%)	140	(2.5%)	8	(2.6%)
Oracle	108	(18%)	77	(26%)	45	(15%)	0	(0%)	4	(1.3%)

Table 2 Comparison of different approaches w.r.t. #timeouts for a cutoff time of 5000 CPU seconds for *Random* ($|I| = 600$, $|S| = 9$), *Crafted* ($|I| = 300$, $|S| = 15$), *Application* ($|I| = 300$, $|S| = 18$) and *3s-Set* ($|I| = 5467$, $|S| = 37$) and 600 seconds for *ASP-Set* ($|I| = 313$, $|S| = 8$).

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 index 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" as done 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 necessary times that have to be added up on each processing unit. The sums of the marked times correspond to $\tau_{\sigma,\pi_u}(i)$ in Equation (6) and (3). Next, we determine the smallest sum of times. Therefore, we iteratively determine 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 begin with, 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 or equal to the sum of times of unit `U+1` and 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 the preceding units (Line 25) or the unit `U+1` (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 some short results from an empirical evaluation. The python implementation of our solver, dubbed *aspeed*, uses the ASP systems [4] of the potassco group [6], namely grounder the *gringo* (3.0.4) and the ASP solver *clasp* (2.0.5). The sets of runtime data (including a list of the solvers and instances) used in this work are freely available online [1].

To provide a thorough empirical evaluation of our approach, we selected five large data sets of runtimes for two prominent and widely studied problems, SAT and ASP. The sets *Random*, *Crafted* and *Application* contain the runtimes taken from the 2011 SAT Competition [18]; the *3s-Set* is the training set of the portfolio SAT solver *3s* [13]; and the ASP instance set (*ASP-Set*) contains runtimes based on different configurations of the highly parametric ASP solver *clasp* [8].

Based on these data sets, we compare sequential *aspeed* and parallel *aspeed* with eight cores (*par 8*) against the best solver in the portfolio (*Single Best*), a uniform distribution of the time slices

over all solvers in the portfolio (*Uniform*), the *Oracle* performance (also called *virtual best solver*) and two SAT solvers: a *ppfolio-like* approach inspired by the single-threaded version of *ppfolio*, where the best three complementary solvers are selected with a uniform distribution of time slices, and *satzilla* [23] based on the results of [24] as a representative of a sequential portfolio-based algorithm selector. To obtain an unbiased evaluation of performance, we used 10-fold cross validation. Table 2 shows the number of timeouts and, in brackets, the corresponding fraction of the instance set; hence, small numbers indicate better performance. In all cases, *aspeed* showed better performance than the *Single Best* solver. *aspeed* performed better than *ppfolio-like* in three out of five settings, namely on *Crafted*, *3s-Set* and *ASP-Set*, and better than *satzilla* in one out of three settings, namely, *Crafted*.

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 [16] on one side, and to work by Huberman, Lukos, and Hogg [12] 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 by Rice [16], as well as by the work on SATzilla, which makes use of regression-based models of running time [22, 23]; work on the use of decision trees and case-base reasoning for selecting bid evaluation algorithms in combinatorial auctions [10, 5]; and work on various machine learning techniques for selecting algorithms for finding maximum probable explanations in Bayes nets in real time [11]. 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.

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 [15]. 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 [12, 9]. *ppfolio*, which demonstrated impressive performance at the 2011 SAT Competition, is a simple procedure that runs between 3 and 5 SAT solver 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 [25] 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* [15] and GASS [20]; 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.

Perhaps most closely related to our approach is the recent work of Kadioglu et al. on algorithm selection and scheduling [13]. 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 [14]. (We note that in the literature on artificially created, 'uniform random' SAT and CSP instances there is some evidence suggesting that at least with the cheaply computable features that can be practically exploited by per-instance algorithm selection approaches this assumption may not hold.) 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, we consider the more general case of parallel solver schedules, while their work is limited to sequential execution of solvers.

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. Compared with earlier model-free and model-based approaches (*ppfolio* and *satzilla*, respectively), our new procedure, *aspeed*, performs very well on SAT and ASP – two widely studied problems for which substantial and sustained effort is being expended in the design and implementation of high-performance solvers.

aspeed is open-source and available online [1]. 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 minimisation 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 run 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 [19].

Acknowledgments

This work was partially funded by the German Science Foundation (DFG) under grant SCHA 550/8-2.

References

1 aspeed. Available at `http://www.cs.uni-potsdam.de/aspeed/`.

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

3 A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

4 F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The third answer set programming competition: Preliminary report of the system competition track. In *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*, pages 388–403. Springer-Verlag, 2011.

5 C. Gebruers, A. Guerri, B. Hnich, and M. Milano. 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*, volume 3011 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2004.

6 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

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

8 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007.

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

10 A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI'04)*, pages 475–479, 2004.

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

12 B. Huberman, R. Lukose, and T. Hogg. An economic approach to hard computational problems. *Science*, 27:51–53, 1997.

13 S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm Selection and Scheduling. In *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 454–469. Springer-Verlag, 2011.

14 S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC – instance-specific algorithm configuration. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10)*, pages 751–756. IOS Press, 2010.

15 E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. 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)*, 2008.

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

17 O. Roussel. Description of ppfolio, 2011.

18 SATComp11. Available at `http://www.cril.univ-artois.fr/SAT11/`.

19 M. Schneider and H. Hoos. Quantifying homogeneity of instance sets for algorithm configuration. In *Proceedings of the Sixth International Conference Learning and Intelligent Optimization (LION'12)*, *Lecture Notes in Computer Science*. Springer-Verlag, 2012.

20 M. Streeter, D. Golovin, and S. Smith. Combining multiple heuristics online. In *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07)*, pages 1197–1203. AAAI Press, 2007.

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

22 L. Xu, H. Hoos, and K. Leyton-Brown. Hierarchical Hardness Models for SAT. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *Lecture Notes in Computer Science*, pages 696–711. Springer-Verlag, 2007.

23 L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

24 L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Detailed SATzilla Results from the Data Analysis Track of the 2011 SAT Competition. Technical report, University of British Columbia, 2011.

25 X. Yun and S. Epstein. Learning algorithm portfolios for parallel execution. In *Proceedings of the Sixth International Conference Learning and Intelligent Optimization (LION'12)*, *Lecture Notes in Computer Science*, Springer-Verlag, 2012.