HAL: A Framework for the Automated Analysis and Design of High-Performance Algorithms

Christopher Nell, Chris Fawcett, Holger H. Hoos, and Kevin Leyton-Brown

University of British Columbia, 2366 Main Mall, Vancouver BC, V6T 1Z4, Canada
{cnell,fawcettc,hoos,kevinlb}@cs.ubc.ca

Abstract. Sophisticated empirical methods drive the development of high-performance solvers for an increasing range of problems from industry and academia. However, automated tools implementing these methods are often difficult to develop and to use. We address this issue with two contributions. First, we develop a formal description of *meta-algorithmic problems* and use it as the basis for an automated algorithm analysis and design framework called the High-performance Algorithm Laboratory. Second, we describe HAL 1.0, an implementation of the core components of this framework that provides support for distributed execution, remote monitoring, data management, and analysis of results. We demonstrate our approach by using HAL 1.0 to conduct a sequence of increasingly complex analysis and design tasks on state-of-the-art solvers for SAT and mixed-integer programming problems.

1 Introduction

Empirical techniques play a crucial role in the design, study, and application of high-performance algorithms for computationally challenging problems. Indeed, state-of-the-art solvers for prominent combinatorial problems, such as propositional satisfiability (SAT) and mixed integer programming (MIP), rely heavily on heuristic mechanisms that have been developed and calibrated based on extensive computational experimentation. Performance assessments of such solvers are also based on empirical techniques, as are comparative analyses of competing solvers for the same problem. Advanced algorithm design techniques based on empirical methods have recently led to substantial improvements in the state of the art for solving many challenging computational problems (see, *e.g.*, [1,2,3]).

Empirical analysis and design techniques are often used in an ad-hoc fashion, relying upon informal experimentation. Furthermore, despite a growing body of literature on advanced empirical methodology, the techniques used in practice are often rather elementary. We believe that this is largely due to the fact that many researchers and practitioners do not have sufficient knowledge of, or easy access to, more sophisticated techniques, and that implementations of these techniques are often difficult to use, if publicly available at all. At the same time, it is clear that much can be gained from the use of advanced empirical techniques.

To address the need for easy access to powerful empirical techniques, we developed HAL, the High-performance Algorithm Laboratory – a computational

C.A. Coello Coello (Ed.): LION 5, LNCS 6683, pp. 600–615, 2011.
© Springer-Verlag Berlin Heidelberg 2011

environment for empirical algorithmics. HAL was conceived to support both the computer-aided design and the empirical analysis of high-performance algorithms, by means of a wide range of ready-to-use, state-of-the-art analysis and design procedures [4]. HAL was also designed to facilitate the development, dissemination, and ultimately wide adoption of novel analysis and design procedures.

By offering standardized, carefully designed procedures for a range of empirical analysis and design tasks, HAL aims to promote best practices and the correct use of advanced empirical methods. In particular, HAL was designed to support the use and development of fully automated procedures for the empirical analysis and design of high-performance algorithms. Since they operate upon algorithms, we refer to these procedures as *meta-algorithmic procedures* (or *meta-algorithms*). Example meta-algorithmic analysis procedures include the characterization of algorithm performance on a set of benchmark instances using a solution cost distribution, as well as the comparison of two algorithms' performance using the Wilcoxon signed-rank test (see, *e.g.*, [5]). Meta-algorithmic design procedures are rapidly gaining prominence and include configuration procedures, such as ParamILS [6,7] and GGA [8], and portfolio builders like SATzilla [9,1].

During the early stages of developing HAL, we realized that appropriately formalized notions of meta-algorithmic procedures, and of the tasks accomplished by these procedures, would provide an ideal foundation for the system. This conceptual basis promotes ease of use, by inducing a natural categorization of analysis and design procedures and by facilitating the use of multiple (or alternative) analysis or design procedures. For example, configuration procedures like ParamILS and GGA solve the same fundamental problem, and with HAL it is easy to conduct analogous (or even parallel) experiments using either of them. Furthermore, HAL's foundation on meta-algorithmic concepts facilitates the combination of various procedures (such as configuration and algorithm selection [10]) and their sequential application (such as configuration followed by comparative performance analysis), as well as the application of analysis or design procedures to other meta-algorithmic procedures (as in the automated configuration of a configurator). Finally, meta-algorithmic concepts form a solid basis for realizing HAL in a convenient and extensible way.

HAL also offers several other features important for work in empirical algorithmics. First, to support large computational experiments, HAL uses a database to collect and manage data related to algorithms, benchmark instances, and experimental results. Second, while HAL can be used on a stand-alone computer, it also supports distributed computation on computer clusters. Third, it allows researchers to archive experiment designs into a single file, including settings, instances, and solvers if unencumbered by license restrictions. Another user can load the file into HAL and replicate exactly the same experiment.

HAL is also designed to facilitate the development and critical assessment of meta-algorithmic procedures. To this end, it is realized as an open environment that is easy to extend, and offers strong support for recurring tasks such as launching, monitoring, and analyzing individual algorithm runs. In short,

HAL allows developers to focus more on building useful and powerful meta-algorithmic procedures and less on the infrastructure required to support them. We hope that this will help to bring about methodological progress in empirical algorithmics, and specifically in the development of novel meta-algorithmic procedures, incorporating contributions from a broad community of researchers and practitioners.

HAL shares some motivation with other systems supporting the empirical study of algorithms. PAVER [11] performs automated performance analysis of optimization software through a web-based interface, but requires that input data be collected by separate invocation of a different tool, and thus is unsuitable for automated techniques that perform concurrent data collection and analysis. EDACC [12] is an experiment management framework which, like HAL, supports distributed execution on compute clusters and centralized data storage, accessed via a unified web interface; unlike HAL, EDACC is focused only on the SAT problem, and more fundamentally does not provide any support for automated meta-algorithmic design procedures. Overall, HAL is the only environment of which we are aware that is designed for the development and application of general-purpose meta-algorithmic analysis and design techniques.

The remainder of this paper is structured as follows. In Section 2, we describe in more detail our vision for HAL and the meta-algorithmic concepts underlying it. In Section 3, we explain how HAL 1.0, our initial implementation of the HAL framework, provides an extensible environment for empirical algorithmics research. We illustrate the use of HAL 1.0 with a sequence of analysis and design tasks for both SAT and MIP in Section 4: first characterizing one solver's performance, next comparing alternative solvers, and finally automating solver design using proven meta-algorithmic techniques. Finally, in Section 5 we summarize our contributions and discuss ongoing work.

2 HAL: A Framework for Meta-algorithmics

The concepts of meta-algorithmic analysis and design procedures are fundamental to HAL. In this section we formally introduce these concepts, discuss benefits we can realize from this formal understanding, and outline HAL's high-level design.

2.1 Meta-algorithmic Problems

We begin by defining a (computational) *problem* as a high-level specification of a relationship between a space of inputs and a corresponding space of outputs. An *instance* of a problem p is any set of values compatible with its input space, and a *solution* to an instance is a set of values compatible with its output space and satisfying the relationship required by p. For example, SAT can be defined as:

Input: $\langle V, \phi \rangle$, where V is a finite set of variables, and ϕ is a Boolean formula in conjunctive normal form containing only variables from V or their negations;

Output: $s = \begin{cases} \text{true} & \text{if } \exists K : V \mapsto \{\text{true}, \text{false}\} \text{ such that } \phi = \text{true under } K; \\ \text{false} & \text{otherwise.} \end{cases}$

Thus, $\langle V = \{a, b, c\}, \phi = (\neg b \vee c) \wedge (a \vee b \vee \neg c)\rangle$ is an example of a SAT instance with solution $s = $ true.

An *algorithm* is any well-defined computational procedure that takes some set of inputs and produces some set of outputs. We say an algorithm A solves a problem p if it accepts any instance of p as a subset of its inputs, and a solution to that instance is identified in its outputs when executed. We observe that A may include inputs and/or outputs other than those required by p, and distinguish three types of algorithm inputs: the algorithm-independent problem instance to be solved, algorithm-specific *parameters* that qualitatively affect behaviour while solving the instance, and any other *settings* that might be required (*e.g.*, a CPU time budget or a random seed). We refer to algorithms that have parameters as *parameterized*, and to the rest as *parameterless*. Any parameterized algorithm can be made parameterless by instantiating all of its parameters with specific values. Thus, a parameterized algorithm defines a space of parameterless algorithms.

A *meta-algorithmic problem* is a problem whose instances contain one or more algorithms, and a meta-algorithm, or *meta-algorithmic procedure*, is an algorithm that solves some meta-algorithmic problem. We refer to algorithms that serve as (part of) a meta-algorithm's input as *target algorithms*, and to the problems target algorithms solve as *target problems*. An *analysis problem* is a meta-algorithmic problem whose solution must include a statement about the target algorithm(s); a *design problem* is a meta-algorithmic problem whose solutions must include one or more algorithms. Finally, we refer to an algorithm that solves an analysis problem as an *analysis procedure*, and one that solves a design problem as a *design procedure*.

Meta-algorithmic analysis problems are ubiquitous, even if they are not always solved by automated procedures. Consider the task of evaluating a solver on a benchmark instance set, using various statistics and diagnostic plots. This corresponds to the *single-algorithm analysis problem*:

Input: $\langle A, \mathcal{I}, m\rangle$, where A is a parameterless target algorithm, $\mathcal{I}$ is a distribution of target problem instances, and m is a performance metric;

Output: $\langle S, T\rangle$, where S is a list of scalars and T a list of plots; and where each $s \in S$ is a statistic describing the performance of A on $\mathcal{I}$ according to m, and each $t \in T$ is a visualization of that performance.

One meta-algorithmic procedure for solving this problem might collect runtime data for A, compute statistics including mean, standard deviation, and quantiles, and plot the solution cost distribution over the instance set [5]; other procedures might produce different plots or statistics. We can similarly define *pairwise comparison*, whose instances contain *two* parameterless algorithms, and whose output characterizes the two algorithms' relative strengths and weaknesses.

Now consider the use of ParamILS [6,7] to optimize the performance of a SAT solver. ParamILS is a meta-algorithmic design procedure that approximately solves the *algorithm configuration problem*:

Input: $\langle A, \mathcal{I}, m \rangle$, where A is a parameterized target algorithm, $\mathcal{I}$ is a distribution of target problem instances, and m is a performance metric;

Output: A^*, a parameterless algorithm for the target problem; where A^* corresponds to an instantiation of A's parameters to values that optimize aggregate performance on $\mathcal{I}$ according to m.

We can similarly define the per-instance *portfolio-based algorithm selection problem*, which is approximately solved by SATzilla [9,1]:

Input: $\langle \mathcal{A}, \mathcal{I}, m \rangle$, where $\mathcal{A}$ is a finite set of parameterless target algorithms, $\mathcal{I}$ is a distribution of target problem instances, and m is a performance metric;

Output: A', a parameterless algorithm for the target problem; where A' executes one $A \in \mathcal{A}$ for each input instance, optimizing performance according to m.

Other variations also fit within the framework. Since we consider a parameterized algorithm to be a space of parameterless algorithms, portfolio-based selection can be seen as a special case of the generalization of configuration sometimes referred to as *per-instance configuration*, restricted to finite sets of target algorithms. Generalizing differently, we can arrive at the *parallel portfolio scheduling problem*, which requires that A' executes multiple algorithms from $\mathcal{A}$ in parallel and returns the first solution found, allocating computational resources to optimize the expected aggregate performance on $\mathcal{I}$ according to m. Finally, one can further generalize to *per-instance parallel portfolio scheduling*, where A' executes multiple algorithms from $\mathcal{A}$ for each input instance and returns the first solution found, allocating computational resources to optimize performance according to m.

We note a parallel between meta-algorithmic problems and the idea of design patterns from software engineering, which describe recurrent problems arising frequently in a given environment, along with solutions for them [13]. Meta-algorithmic problems identify challenges that arise regularly in algorithm development and present specific solutions to those challenges. However, choosing between design patterns relies on understanding the benefits and drawbacks of each. The same holds in the meta-algorithmic context; we hope that HAL will prove useful for developing such understanding.

2.2 The High-Performance Algorithm Laboratory

HAL has been designed to align closely with the conceptual formalization from Section 2.1, thereby providing a unified environment for the empirical analysis and design of high-performance algorithms via general meta-algorithmic techniques. In particular, HAL allows explicit representation of arbitrary problems and algorithms (including input and output spaces), problem instances and distributions, and performance metrics. Meta-algorithmic problems in HAL are simply problems whose input (and perhaps output) spaces are constrained to involve algorithms; likewise, meta-algorithmic procedures are realized as a special case of algorithms. HAL presents a unified user interface that gives the user easy and uniform access to a wide range of empirical analysis and design techniques through a task-basked workflow. For example, users can design experiments

simply by selecting a meta-algorithmic problem of interest (*e.g.*, configuration), a meta-algorithmic procedure (*e.g.*, ParamILS), and additional information that specifies the meta-algorithmic problem instance to be solved (*e.g.*, a target algorithm, a distribution of target instances, and a performance metric).

This design provides the basis for five desirable characteristics of HAL. First, it allows HAL to work with arbitrary problems, algorithms and meta-algorithmic design and analysis techniques. Second, it enables HAL to automatically archive and reuse experimental data (avoiding duplication of computational effort, *e.g.*, when rerunning an experiment to fill in missing data), and to serve as a central repository for algorithms and instance distributions. Third, it makes it easy to support packaging and distribution of complete experiments (including target algorithms, instances, and other experiment settings) for independent verification, for example to accompany a publication. Fourth, it facilitates the straightforward use (and, indeed, implementation) of different meta-algorithmic procedures with compatible input spaces; in particular including procedures that solve the same meta-algorithmic problem (*e.g.*, two algorithm configuration procedures). Finally, it simplifies the construction of complex experiments consisting of sequences of distinct design and analysis phases.

To support a wide range of meta-algorithmic design and analysis procedures, HAL allows developers to contribute self-contained plug-in modules relating to specific meta-algorithmic problems and their associated procedures. A plug-in might provide a new procedure for a relatively well-studied problem, such as configuration. Alternately, it might address new problems, such as robustness analysis or algorithm simplification, and procedures for solving them drawing on concepts such as solution cost and quality distributions, runtime distributions, or parameter response curves. In the long run, the value of HAL to end users will largely derive from the availability of a library of plug-ins corresponding to cutting-edge meta-algorithmic procedures. Thus, HAL is an open platform, and we encourage members of the community to contribute new procedures.

To facilitate this collaborative approach, HAL is designed to ensure that the features offered to end users are mirrored by benefits to developers. Perhaps most importantly, the separation of experiment design from runtime details means that the execution and data management features of HAL are automatically provided to all meta-algorithmic procedures that implement the HAL API. The API also includes implementations of the fundamental objects required when building a meta-algorithm, and makes it easier for developers to implement new meta-algorithmic procedures. Adoption of this standardized API also streamlines the process of designing hybrid or higher-order procedures. For example, both Hydra [10] and ISAC [14] solve algorithm configuration and per-instance portfolio-based selection problems; implementation using HAL would allow the underlying configuration and selection sub-procedures to be easily replaced or interchanged. Finally, as we continue to add meta-algorithmic procedures to HAL, we will compile a library of additional functionality useful for implementing design and analysis procedures. We expect this library to ultimately include components for exploring design spaces (*e.g.*, local search and continuous optimization), machine learning (*e.g.*, feature extraction and regression/classification

methods), and empirical analysis (*e.g.*, hypothesis testing and plotting), adapted specifically for the instance and runtime data common in algorithm design scenarios.

3 The HAL 1.0 Core Infrastructure

The remainder of this paper describes an implementation of HAL's core functionality, HAL 1.0, which is now available online.[1] The system is essentially complete in terms of core infrastructure (*i.e.*, experiment modelling, execution management, and user interface subsystems), and includes five meta-algorithmic procedures, focused on two meta-algorithmic analysis problems—single algorithm analysis and paired comparison—and the meta-algorithmic design problem of configuration. These procedures are further described in Section 4, where we present a case study illustrating their use. As discussed above, we intend to add a variety of additional meta-algorithmic procedures to HAL in the next release, and hope that still others will be contributed by the broader community.

This section describes HAL 1.0's core infrastructure. We implemented HAL 1.0 in Java, because the language is platform independent and widely used, its object orientation is appropriate for our modular design goals, and it offers relatively high performance. The HAL 1.0 server has been tested primarily under openSUSE Linux and Mac OS X, and supports most POSIX-compliant operating systems; basic Windows support is also provided. The web-based UI can provide client access to HAL from any platform. HAL 1.0 interfaces with Gnuplot for plotting functionality, and (optionally) with R for statistical computing (otherwise, internal statistical routines are used), MySQL for data management (otherwise, an embedded database is used), and Grid Engine for cluster computing.

In the following subsections, we describe HAL 1.0's implementation in terms of the three major subsystems illustrated in Figure 1. While these details are important for prospective meta-algorithm contributors and illustrative to readers in general, one does not need to know them to make effective use of HAL 1.0.

3.1 Experiment Modelling

The components of the *experiment modelling subsystem* correspond to the concepts defined in Section 2. This subsystem includes most of the classes exposed to developers using the HAL API, including those that are extensible via plug-ins.

We will consider the running example of a user designing an experiment with HAL 1.0, which allows us to describe the Java classes in each subsystem. The user's first step is to select a meta-algorithmic problem to solve. The *Problem* class in HAL 1.0 encodes the input and output *Spaces* defined by a particular computational problem. (We hereafter indicate Java classes by capitalizing and italicizing their names.) The relationship between the inputs and outputs is not explicitly encoded, but is implicitly identified through the name of the

[1] `hal.cs.ubc.ca`

Problem itself. Individual variables in a *Space* are represented by named *Domains*; functionality is provided to indicate the semantics of, and conditional interdependencies between, different variables. HAL 1.0 supports a variety of *Domains*, including Boolean-, integer-, and real-valued numerical *Domains*, categorical *Domains*, and *Domains* of other HAL objects.

Once a problem is selected, the user must import an *InstanceDistribution* containing target problem *Instances* of interest. HAL 1.0 currently supports finite instance lists, but has been designed to allow other kinds of instance distributions such as instance generators. The *Instance* class provides access to problem-specific instance data, as well as to arbitrary sets of *Features* and user-provided *Tags* (used, *e.g.*, to indicate encoding formats that establish compatibility with particular *Problems* or *Algorithms*). An *Instance* of a target problem typically includes a reference to the underlying instance file; an *Instance* of a meta-algorithmic problem contains the *Algorithms*, *Instances*, and *Metrics* that define it.

The next step in experiment specification is to choose one or more target algorithms. In HAL 1.0, the *Algorithm* class encodes a description of the input and output spaces of a particular algorithm *Implementation.* For external target algorithms, the *Implementation* specifies how the underlying executable is invoked, and how outputs should be parsed; for meta-algorithmic procedures, it implements the relevant meta-algorithmic logic. Note that the base *Implementation* classes are interfaces, and meta-algorithmic procedures added via plug-ins provide concrete implementations of these. Input and output spaces are encoded using the *Space* class, and an *Algorithm* may be associated with a set of *Tags* that identify the *Problems* that the algorithm solves, and compatible *Instances* thereof. Two *Algorithm* subclasses exist: a *ParameterizedAlgorithm* includes configurable parameters in its input space, and a *ParameterlessAlgorithm* does not. Before execution, an *Algorithm* must be associated with a compatible *Instance* as well as with *Settings* mapping any other input variables to specific values.

The final component needed to model a meta-algorithmic experiment is a performance metric. A *Metric* in HAL 1.0 is capable of performing two basic actions: first, it can evaluate an *AlgorithmRun* (see Section 3.2) to produce a single real value; second, it can aggregate a collection of such values (for example, over problem instances, or over separate runs of a randomized algorithm) into a single final score. HAL 1.0 includes implementations for commonly-used performance metrics including median, average, penalized average runtime (PAR), and average solution quality, and it is straightforward to add others as required.

3.2 Execution and Data Management

The *execution subsystem* implements functionality for conducting experiments specified by the user; in HAL 1.0, it supports execution on a local system, on a remote system, or on a compute cluster. It also implements functionality for cataloguing individual resources (such as target algorithms or instance distributions) and for archiving and retrieving the results of runs from a database.

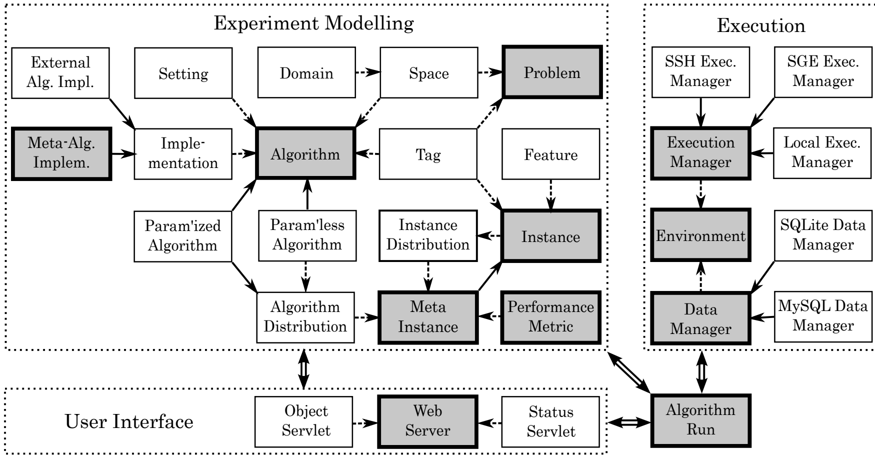

Fig. 1. Infrastructural overview of HAL 1.0. Dashed arrows indicate composition; solid arrows, inheritance. Key components are shaded. Note the distinct subsystems, with interactions between them (double arrows) typically moderated by *AlgorithmRuns*.

Once our user has completely specified an experiment, he must define the environment in which execution is to occur. An *Environment* in HAL 1.0 is defined by *ExecutionManagers* which are responsible for starting and monitoring computation and a *DataManager* which is responsible for performing archival functions. When an algorithm run request is made, the *Environment* queries the *DataManager* to see if results for the run are already available. If so, these results are fetched and returned; if not, the request is passed to an *ExecutionManager* for computation and automatic output parsing. In either case, results are returned as an *AlgorithmRun* object which allows monitoring of the run's elapsed CPU time, status, and individual output value trajectories both in real time during execution and after completion. It also exposes functionality for early termination of runs and uses this to enforce runtime caps.

HAL 1.0 includes three *ExecutionManager* implementations. The *LocalExecutionManager* performs runs using the same machine that runs HAL 1.0, and the *SSHExecutionManager* performs runs on remote machines using a secure shell connection. The *SGEClusterExecutionManager* distributes algorithm runs to nodes of a compute cluster managed by Oracle Grid Engine (formerly Sun Grid Engine). The *Environment* can be configured to use different *ExecutionManagers* in different situations. For example, for analysis of an algorithm on target problems that require a particularly long time to solve, the user might specify an *Environment* in which the parent meta-algorithm is executed on the local machine, but target algorithm runs are distributed on a cluster. Alternatively, when target algorithm runs are relatively short but require a platform different than the one running HAL 1.0, the user might specify an *Environment* in which all execution happens on a single remote host.

HAL 1.0 includes two *DataManager* implementations. By default, a subclass employing an embedded SQLite database is used. However, due to limitations

of SQLite in high-concurrency applications, a MySQL-backed implementation is also provided. These *DataManagers* use a common SQL schema based on the same set of fundamental meta-algorithmic concepts to store not only experimental results, but also information sufficient to reconstruct all HAL objects used in the context of a computational experiment. We note that external problem instances and algorithms are not directly stored in the database, but instead at recorded locations on the file system, along with integrity-verifying checksums. This eliminates the need to copy potentially large data files for every run, but presently requires that all compute nodes have access to a shared file space.

3.3 User Interface

The *user interface subsystem* provides a remotely-accessible web interface to HAL 1.0, via an integrated *WebServer*. Many classes have associated *ObjectServlets* in the *WebServer*, which provide interface elements for their instantiation and modification. The *ObjectServlets* corresponding to *Problems* are used to design and execute experiments; the servlet for a given *Problem* automatically makes available all applicable meta-algorithmic procedures. Additional *ObjectServlets* allow the user to specify and examine objects such as *Algorithms*, *InstanceDistributions*, *Settings*, and *Environments*. A *StatusServlet* allows the user to monitor the progress and outputs of experiments both during and after execution, by inspecting the associated *AlgorithmRun* objects. Finally, the interface allows the user to browse and maintain all objects previously defined in HAL, as well as to export these objects for subsequent import by other users.

4 Case Study: Analysis and Design with HAL 1.0

We now demonstrate HAL 1.0 in action. Specifically, we walk through two workflow scenarios that could arise for a typical user. In this way, we also present the five meta-algorithmic procedures that are available in HAL 1.0. The outputs of these procedures are summarized in Table 1, and in the following figures (exported directly from HAL 1.0). Exports of experiment designs are available on the HAL website to facilitate independent validation of our findings.

Scenario 1: Selecting a MIP Solver. In this scenario, a user wants to select between two commercial mixed-integer program (MIP) solvers, IBM ILOG CPLEX[2] 12.1 and GUROBI[3] 3.01, on the 55-instance mixed integer linear programming (MILP) benchmark suite constructed by Hans Mittelmann.[4] Our user sets a per-target-run cutoff of 2h and uses penalized average runtime (PAR-10) as the performance metric (PAR-k counts unsuccessful runs at k times the cutoff).

Scenario 2: Adapting a SAT Solver. In this scenario, a user aims to adapt a stochastic tree search solver for SAT, version 1.2.1 of SPEAR [15], to achieve

[2] `ibm.com/software/integration/optimization/cplex`
[3] `gurobi.com`
[4] `plato.asu.edu/ftp/milpf.html`

strong performance on the 302-instance industrial software verification (SWV) benchmark training and test sets used by Hutter et al. [16]. Our user sets a per-target-run cutoff of 30s and evaluates performance by mean runtime (PAR-1).

Computational Environment. All experiments were performed on a Grid Engine cluster of 55 identical dual-processor Intel Xeon 3.2GHz nodes with 2MB cache and 4GB RAM running openSUSE Linux 11.1. Runtime data was archived using a dedicated MySQL server with the same machine specifications. Individual target algorithm runs for Scenario 1 experiments were distributed across cluster nodes, and for Scenario 2 experiments were consolidated on single nodes. Reported runtimes indicate CPU time used, as measured by HAL 1.0.

4.1 The Single-Algorithm Analysis Problem

In both scenarios, our user begins by analyzing single algorithms individually.

Analysis Procedure 1: SCD-Based Analysis. This comprehensive approach to single-algorithm analysis takes as input a single target algorithm, a set of benchmark instances, and some additional settings including a maximum number of runs per target instance, a maximum CPU time per target run, a maximum number of total target runs, and a maximum aggregate runtime budget. It collects runtime data for the target algorithm on the instance distribution (in parallel, when specified) until a stopping criterion is satisfied. Summary statistics are computed over the instance distribution, and a solution cost distribution plot (SCD; see, *e.g.*, Ch. 4 of [5]), illustrating (median) performance across all target runs on each instance, is produced.

Scenario 1(1). CPLEX is the most prominent mixed-integer programming solver. Here, our user measures its performance on the MILP instance set using the SCD-Based Analysis procedure; as CPLEX is deterministic, it is run only once per instance. The resulting summary statistics are shown in Table 1, and the SCD appears in the left pane of Figure 2.

Scenario 2(1). SPEAR was originally optimized for solving SAT instances from several applications, but was later prominently used for software verification in particular. In this phase of the case study, our user assesses the original, manually optimized version of SPEAR on the SWV test set. The summary statistics from an SCD-based analysis (performing 20 runs per instance as SPEAR is randomized) are shown in Table 1 and the SCD in the top left pane of Figure 3.

4.2 The Pairwise Comparison Problem

Now our user performs pairwise comparisons between different solvers.

Analysis Procedure 2: Comprehensive Pairwise Comparison. This procedure performs SCD-Based Analysis on two given algorithms, generates a scatter plot illustrating paired performance across the given instance set, and performs Wilcoxon signed-rank and Spearman rank correlation tests. The Wilcoxon signed-rank test determines whether the median of the paired

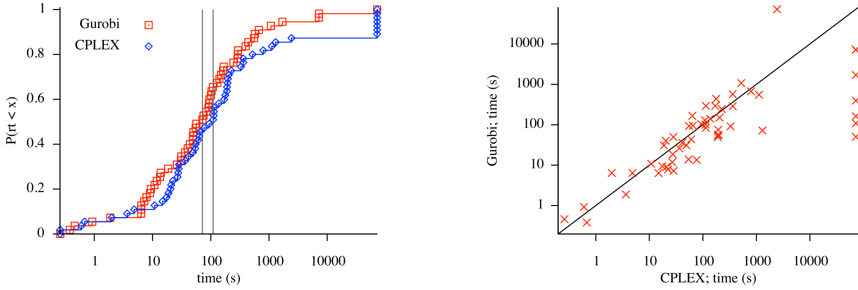

Fig. 2. Comparison of CPLEX and GUROBI on the MILP benchmark set. In the SCD, median runtimes are indicated by vertical lines.

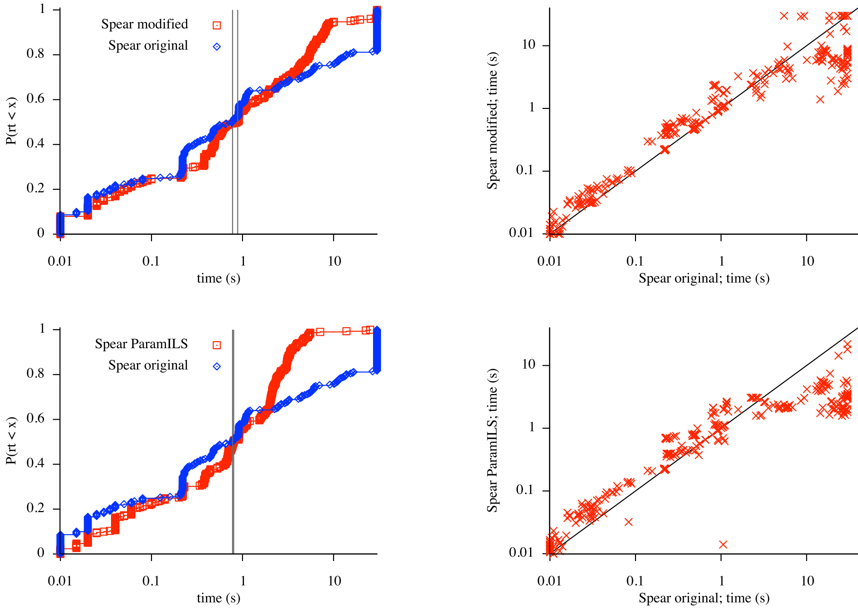

Fig. 3. Analysis of SPEAR designs on SWV test set. Top row, original vs. intuitively modified design; bottom row, original vs. best configured design (from PARAMILS).

performance differences between the two algorithms across the instance set is significantly different from zero; if so, it identifies the better-performing algorithm. The Spearman rank correlation test determines whether a significant monotonic performance correlation exists between them. Both tests are non-parametric, and so appropriate for the non-Gaussian performance data frequently seen in empirical algorithm analysis.

Scenario 1(2). Our user aims to compare CPLEX with GUROBI, a relatively recent commercial MIP solver. He uses HAL's Comprehensive Pairwise Comparison procedure on the MILP benchmark set for this task. Statistics on the performance of the two solvers are shown in Table 1. As can be seen from Figure 2, which presents the combined SCD plot and the performance correlation

plot, GUROBI outperformed CPLEX on most instances; the Wilcoxon signed-rank test indicated that this performance difference was significant at $\alpha = 0.05$ ($p = 0.024$). This result is consistent with Mittelmann's observations using the MILP benchmark set. A Spearman correlation coefficient of $\rho = 0.86$ ($p = 0.0$) reflects the strong correlation seen in the scatter plot. However, the slightly better performance of CPLEX observed for a number of instances suggests a potential for modest performance gains by using automated portfolio-based algorithm selection techniques (see, *e.g.*, [1]), which we plan to support in HAL in the near future.

Scenario 2(2). When adapting an algorithm to a new class of benchmark instances, algorithm designers often apply intuition to making important design choices; these choices are often realized by setting parameters of the algorithm to certain values. For example, Hutter et al. [16] provide an intuitive explanation of the strong performance of one particular configuration of SPEAR in solving software verification instances. Our user follows their qualitative description to manually obtain a configuration of SPEAR that he then compares against the default on the SWV test set (based on 20 runs per instance) using the Comprehensive Pairwise Comparison procedure; the results are shown in Figure 3 and Table 1. Overall, the modified configuration achieved better (PAR-1) performance than the default, as expected. However, as clearly seen from the SCDs and from the scatter plot, this was accomplished by sacrificing performance on easy instances for gains on hard instances. The Wilcoxon signed-rank test determined that if all instances were weighted equally, the median paired performance difference over the full benchmark set was *not* significantly different from zero at $\alpha = 0.05$ ($p = 0.35$). The inter-instance correlation was significant, however, with $\rho = 0.97$ ($p = 0.0$).

4.3 The Algorithm Configuration Problem

In Scenario 2(2) above, our user observed that SPEAR's performance can be improved by manually modifying its parameters. Seeking further performance gains, he turns to automatic configuration. HAL 1.0 supports three procedures for this meta-algorithmic design problem.

Design Procedure 1: Automated Configuration using ParamILS. HAL 1.0 supports the FOCUSEDILS variant of the local-search-based PARAMILS configurator [7]. The original Ruby implementation is augmented by using an adapter class to implement the plugin in HAL 1.0.

Design Procedure 2: Automated Configuration using GGA. HAL 1.0 includes a plugin that interfaces with the original implementation of GGA, which employs a gender-based genetic algorithm [8]. Unfortunately, sources for this procedure are not available, and because of copyright restrictions we are unable to further distribute the executable supplied to us by its authors.

Design Procedure 3: Automated Configuration using ROAR. HAL 1.0 also supports the Random Online Aggressive Racing (ROAR) procedure, a simple yet powerful model-free implementation of the general Sequential Model-Based

Table 1. Summary of case study results. Reported statistics are in terms of PAR-10 for CPLEX and GUROBI, and PAR-1 for SPEAR; units are CPU seconds. Only the best design in terms of training set performance is reported for each configuration procedure.

	Training Set					Test Set				
Algorithm	q25	q50	q75	mean	stddev	q25	q50	q75	mean	stddev
CPLEX						26.87	109.93	360.59	9349.1	24148.9
GUROBI						13.45	71.87	244.81	1728.8	9746.0
SPEAR default						0.13	0.80	10.78	6.78	10.62
SPEAR modified						0.19	0.89	4.35	3.40	6.31
SPEAR PARAMILS	0.22	0.80	2.63	1.72	2.54	0.19	0.80	2.21	1.56	2.22
SPEAR GGA	0.22	0.90	1.96	2.00	3.36	0.16	0.90	1.72	1.72	3.39
SPEAR ROAR	0.22	0.92	2.70	1.91	2.59	0.19	0.91	2.41	1.82	2.98

Optimization (SMBO) framework [17]. ROAR was implemented entirely within HAL 1.0, and serves as an example of developing meta-algorithmic design procedures within the HAL framework.

Unlike ROAR and GGA, PARAMILS requires sets of discrete values for all target algorithm parameters; therefore, when using PARAMILS, HAL 1.0 automatically discretizes continuous parameters. Unlike PARAMILS and ROAR, GGA requires all target runs to be performed on the same host machine, and GGA's authors recommend against the use of performance metrics other than average runtime.

Scenario 2(3) Because the three configuration procedures are easily interchangeable in HAL 1.0, our user runs all of them. He performs 10 independent runs of each configurator on the SWV training set, and sets a time budget of 3 CPU days for each run. For some of SPEAR's continuous parameters, our user indicates that a log transformation is appropriate. In these cases, HAL performs the transformations automatically when calling each configurator; it also automatically discretizes parameters for ParamILS. Our user validates the performance of each of the 30 final designs on the training set using the SCD-Based Analysis procedure with 20 runs per instance. He then compares the design with the best training performance found by each of the procedures against the default configuration using the Comprehensive Pairwise Comparison procedure on the test set, again performing 20 runs per instance. Results are shown in Figure 3 and Table 1. The best design found by each configurator was substantially better than both the default and the intuitively-modified configuration in terms of PAR-1, with PARAMILS producing slightly better results than GGA, and with GGA in turn slightly better than ROAR. In all cases, the performance difference with respect to the default was significant at $\alpha = 0.05$ according to the Wilcoxon signed rank test ($p = \{7.8, 9.7, 0.002\} \times 10^{-3}$ for PARAMILS, ROAR, and GGA respectively).

5 Conclusions and Future Work

In this work we introduced HAL, a versatile and extensible environment for empirical algorithmics, built on a novel conceptual framework that formalizes meta-algorithmic problems and procedures. HAL facilitates the application of

advanced empirical methods, including computationally intensive analysis and design tasks. It also supports the development and critical assessment of novel empirical analysis and design procedures. The first implementation of our framework, HAL 1.0, can address arbitrary target problems; can run experiments on local machines, remote machines, or distributed clusters; and offers detailed experiment monitoring and control, both before, during and after execution. HAL 1.0 provides a versatile API for developing and deploying new meta-algorithmic analysis and design procedures. Using this API, we developed plugins implementing two performance analysis tasks and supporting three state-of-the-art automated algorithm configurators. We demonstrated the use of all five procedures in a case study involving prominent solvers for MIP and SAT.

Our group continues to actively develop and extend the HAL framework. We are currently working on adding support for additional meta-algorithmic design procedures, such as SATzilla [1], the Hydra instance-based portfolio-builder [10], and the Sequential Model-Based Optimization framework [17]. We are also working on adding new analysis procedures, such as comparative analysis of more than two algorithms and scaling analyses. Finally, we plan to improve HAL's support for execution of experiments on Windows platforms, and on computer clusters running Torque. Ultimately, we hope that the HAL software framework will help to promote the use of state-of-the-art methods and best practices in empirical algorithmics, and to improve the state of the art in solving challenging computational problems through the use of advanced empirical techniques.

Acknowledgements. We thank Frank Hutter for partly implementing Roar and testing HAL, and Meinolf Sellmann and Kevin Tierney for their support in using GGA. Our research has been funded by the MITACS NCE program, by individual NSERC Discovery Grants held by HH and KLB, and by an NSERC CGS M held by CN.

References

1. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. JAIR 32, 565–606 (2008)
2. Chiarandini, M., Fawcett, C., Hoos, H.H.: A modular multiphase heuristic solver for post enrollment course timetabling (extended abstract). In: PATAT (2008)
3. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Automated configuration of mixed integer programming solvers. In: Lodi, A., Milano, M., Toth, P. (eds.) CPAIOR 2010. LNCS, vol. 6140, pp. 186–202. Springer, Heidelberg (2010)
4. Hoos, H.H.: Computer-aided design of high-performance algorithms. Technical Report TR-2008-16, University of British Columbia, Computer Science (2008)
5. Hoos, H.H., Stützle, T.: Stochastic Local Search—Foundations and Applications. Morgan Kaufmann Publishers, USA (2004)
6. Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: AAAI (2007)
7. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: An automatic algorithm configuration framework. JAIR 36, 267–306 (2009)

8. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 142–157. Springer, Heidelberg (2009)
9. Nudelman, E., Leyton-Brown, K., Devkar, A., Shoham, Y., Hoos, H.H.: Understanding random SAT: Beyond the clauses-to-variables ratio. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 438–452. Springer, Heidelberg (2004)
10. Xu, L., Hoos, H.H., Leyton-Brown, K.: Hydra: Automatically configuring algorithms for portfolio-based selection. In: AAAI (2010)
11. Mittelmann, H.D., Pruessner, A.: A server for automated performance analysis of benchmarking data. Opt. Meth. Soft. 21(1), 105–120 (2006)
12. Balint, A., Gall, D., Kapler, G., Retz, R.: Experiment design and administration for computer clusters for SAT-solvers (EDACC). JSAT 7, 77–82 (2010)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, New York (1995)
14. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC – Instance-specific algorithm configuration. In: ECAI (2010)
15. Babić, D.: Exploiting Structure for Scalable Software Verification. PhD thesis, University of British Columbia, Vancouver, Canada (2008)
16. Hutter, F., Babić, D., Hoos, H.H., Hu, A.: Boosting verification by automatic tuning of decision procedures. In: FMCAD (2007)
17. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration (extended version). Technical Report TR-2010-10, University of British Columbia, Computer Science (2010)