

contributed articles

DOI:10.1145/2076450.2076469

Avoid premature commitment, seek design alternatives, and automatically generate performance-optimized software.

BY HOLGER H. HOOS

Programming by Optimization

WHEN CREATING SOFTWARE, developers usually explore different ways of achieving certain tasks. These alternatives are often eliminated or abandoned early in the process, based on the idea that the flexibility they afford would be difficult or impossible to exploit later. This article challenges this view, advocating an approach that encourages developers to not only avoid premature commitment to certain design choices but to actively develop promising alternatives for parts of the design. In this approach, dubbed Programming by Optimization, or PbO, developers specify a potentially large design space of programs that accomplish a given task, from which versions of the program optimized for various use contexts are generated automatically, including parallel versions derived from the same sequential sources. We outline a simple, generic programming language extension that supports the specification of such design spaces and discuss ways specific programs

that perform well in a given use context can be obtained from these specifications through relatively simple source-code transformations and powerful design-optimization methods. Using PbO, human experts can focus on the creative task of devising possible mechanisms for solving given problems or subproblems, while the tedious task of determining what works best in a given use context is performed automatically, substituting human labor by computation.

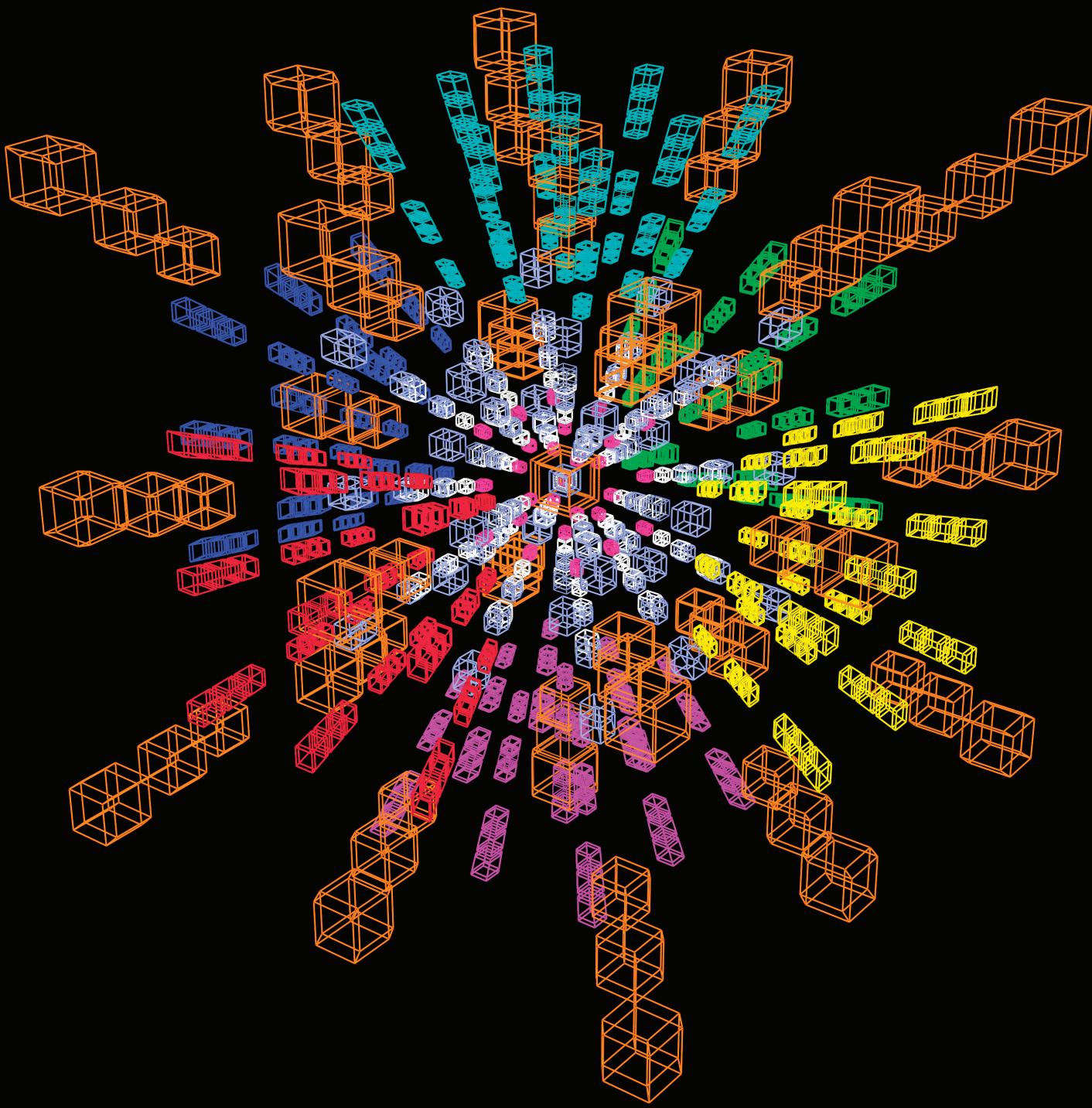
The potential of PbO is evident from recent empirical results (see the table here). In the first two use cases—mixed integer programming and planning—existing software exposing many design choices in the form of parameters was automatically optimized for speed. This resulted in, for example, up to 52-fold speedups for the widely used commercial IBM ILOG CPLEX Optimizer software for solving mixed-integer programming problems.²¹ In the third use case—verification problems encoded into propositional satisfiability—the proactive development of alternatives for important components of the program were an important part of the design process, enabling even greater performance gains.

Performance Matters

Computer programs and the algo-

» key insights

- Premature commitment to design choices during software development often leads to loss of performance and limited flexibility.
- PbO aims to avoid premature design choices and actively develop design alternatives, leading to large and rich design spaces of programs that can be specified through simple generic extensions of existing programming languages.
- Advanced optimization and machine-learning techniques make it possible to perform automated performance optimization over the large spaces of programs arising in PbO-based software development; per-instance algorithm selectors and parallel algorithm portfolios can be obtained from the same sequential source.



MagicCube5D, a fully functional five-dimensional analogue of Rubik's Cube.

rithms on which they are based frequently involve different ways of getting something done. Sometimes, certain choices are clearly preferable, but it is often unclear *a priori* which of several design decisions will ultimately give the best results. Such design choices can, and, routinely, do, occur at many levels, from high-level architectural aspects of a software system to low-level implementation details. They are often made based on consid-

erations of maintainability, extensibility, and performance of the system or program under development. This article focuses on this latter aspect of a system's performance, considering only sets of semantically equivalent design choices and situations in which the performance of a program depends on the decisions made for each part of the program for which one or more candidate designs are available, even though these choices do not

affect the program's correctness and functionality. Note this premise differs fundamentally from that of program synthesis, in which the primary goal is to come up with a design that satisfies a given functional specification.

It may appear that (partly due to the sustained, exponential improvement in computer hardware over more than five decades) software performance is a relatively minor concern. However, upon closer inspection this is far from

Speedups achieved through PbO in conjunction with an automated configurator^{19,20} for performance optimization of solvers for three prominent NP-hard problems in various application contexts; these speedups are with respect to default configurations determined by human experts based on substantial manual effort, and both ends of the ranges shown refer to averages over large sets of benchmark instances.

Problem	Solver	# Parameters	# Configurations	Speedup	Reference
Mixed integer programming	CPLEX	76	1.9×10^{47}	2–52 ×	Hutter et al. ²¹
Planning	LPG	62	6.5×10^{17}	3–118 ×	Vallati et al. ³⁵
Propositional satisfisfiability hardware and software verification	SPEAR	26	8.3×10^{17}	3–525 ×	Hutter et al. ¹⁸

true. Problems that are **NP-hard** and considered computationally intractable are at the heart of a range of challenging tasks encountered in practical applications of considerable importance for the worldwide economy, including scheduling, time-tabling, resource allocation, production planning and optimization, computer-aided design, and software verification.

We expect that, as economic constraints tighten, finding good solutions to these problems will, in many cases, become more difficult. For example, resource-allocation problems are typically easy to solve if there is an abundance of resources relative to the demands in a given situation. Conversely, as demands substantially exceed available resources, no allocation will satisfy all of them, and, slightly less obvious, this fact is typically easy to demonstrate. It is between these extremes that the difficult cases arise, where the demands and available resources are balanced enough that finding a satisfactory allocation or demonstrating that none exists becomes computationally difficult.^a

A natural tendency toward this critically constrained, computationally difficult case can be expected in many real-world contexts. The under-constrained case is typically economically wasteful, providing an incentive for increasing demand on resources by, say, enlarging the customer base, taking on more projects, or reducing availability of resources (such as by scaling back personnel or equipment allotment). On the other hand,

the overconstrained case typically corresponds to lost market opportunity and can cause substantial strain within an organization, providing an incentive to increase resource availability. Furthermore, growing awareness and concern about the limitations of natural resources (such as oil and natural gas), along with increased competition in larger markets and just-in-time delivery of goods and services, provide further incentives to find solutions to computationally difficult problems as quickly as possible. That is why the performance of algorithms, and of the software based on them, matters.

Premature Design Choices

In most (if not all) cases, the key to solving computationally challenging problems lies in a combination of design choices, with effects on performance often interacting in complex, unexpected ways. These choices are typically heuristic in the sense that their efficacy can be demonstrated empirically yet remains inaccessible to the analytical techniques used for proving theoretical complexity results. In some cases, choosing among design alternatives is made at development stages preceding the generation of actual code; in others, design decisions have far-reaching effects on other choices, when, say, deciding on higher-level architectural aspects of a system or on specific data structures widely used within a larger piece of software. However, one of several design alternatives is often chosen at or after the implementation stage, and such a choice, while not constraining other parts of the system, may have a substantial effect on overall performance.

Sometimes, decisions of the latter type are deferred to a post-implementation stage and left to the user by exposing them as parameters.^b More often, however, they are hard-coded, either by means of constants within a program or module or by retaining some pieces of code while abandoning alternatives. Especially when implementing heuristic mechanisms, programmers usually make these design choices based on intuition, experience, and perhaps some ad hoc experimentation.

What PbO Means

Experience in designing high-performance heuristic solvers for **NP-hard** problems shows that building software this way leads to suboptimal results in terms of performance and adaptability to different use contexts. Furthermore, considering preliminary evidence from application areas ranging from numerical computation to sorting algorithms, similar concerns arise when tackling polynomial-time computational problems. We therefore advocate an approach in which many design choices are deliberately left open by means of retaining alternative realizations of components or mechanisms and by exposing a large number of parameters.^{15,18,24} These choices are then made by means of running a meta-algorithmic optimization procedure, optimizing the empirical performance obtained in a given use context. Such a use context is characterised by a set (or distribution) of inputs representative of those encountered in a situation in which a given program is used.

The PbO approach is based on the idea of avoiding premature commitment to certain design choices and actively developing promising alternatives for parts of the design. Rather than build a single program for a given purpose, software developers specify a rich and potentially large design space of programs. From this specification, programs that perform well in a given use context are generated automatically through powerful optimization techniques.

PbO allows human experts to focus

^a This argument is closely related to the notion of “critical constrainedness,” as described by Cheeseman et al.⁶

^b Many users, especially those lacking deep insight into the program or system under consideration, tend to keep these parameters at their default values.

on the creative task of imagining possible mechanisms for solving given problems or subproblems, while the tedious job of determining what works best in a given use context is performed automatically, substituting human labor with computation. More complex designs (such as per-instance selectors^{11,25,40} and parallel portfolios^{10,17}) can be generated automatically from the same design-space specification (and sources; see Figure 1). Such designs are increasingly relevant, since they achieve high performance across a range of use contexts.

Influence on Software Development and Deployment

In 2007, our group at the University of British Columbia first employed, unwittingly, the key idea behind PbO in the context of collaborative work on SAT-based software verification.¹⁸ Using off-the-shelf solvers for the propositional satisfiability problem (SAT) has become a standard approach for formally verifying hardware and software. In the project, the idea was to produce a SAT solver that would be especially well suited for dealing with SAT instances produced by a specific static checker, CALYSTO.² In initial stages of the work (carried out by Domagoj Babić and Alan Hu of the ISD Laboratory in the computer science department), a new SAT solver dubbed SPEAR was developed, including a range of techniques from the SAT literature. Because it was unclear which combination of techniques would be most effective for solving the SAT instances produced by CALYSTO, the initial version of SPEAR could be configured flexibly through parameters exposed to the user. On the other hand, finding settings for these parameters that would result in good solver performance on the instances of interest proved challenging, even for its primary designer, Babić. Therefore, the team decided to use the automated algorithm configuration procedure ParamILS (described later)²⁰ to accomplish the task.

What happened next can be seen as the genesis of the PbO paradigm: After seeing how much better the configurations found by ParamILS performed than his manually tuned default settings, Babić decided to expose additional design choices. Some of them

had been previously hardwired into the program; others had been implemented, tested, and then abandoned; yet others were newly implemented alternatives to existing mechanisms within SPEAR. This ultimately led to a version of SPEAR that could be configured via 26 parameters, jointly giving rise to $8.34 \cdot 10^{17}$ configurations of the solver.^c

ParamILS turned out to be able to achieve speedups of a factor of more than 500 for the software-verification instances produced by CALYSTO compared to the default configuration of SPEAR that had been manually determined with considerable effort by its designer.¹⁸ This automatically optimized configuration of SPEAR won the QF_BV category of the 2007 Satisfiability Modulo Theories Competition (<http://www.smtcomp.org/2007/>), using several components, including clause and variable elimination mechanisms, that appeared to be ineffective during earlier, manual-configuration attempts. Moreover, the same highly parametric version of SPEAR, when automatically optimized for solving SAT-encoded hardware verification instances, achieved substantial speedups over the (then) state-of-the-art solver MiniSAT

2.0.¹⁸ Comparing the configurations specifically optimized for SAT-encoded hardware and software verification produced a number of interesting observations; for example, the configuration optimized for software verification was found to use a more aggressive restart mechanism and a simpler phase-selection heuristic than the one optimized for hardware verification.

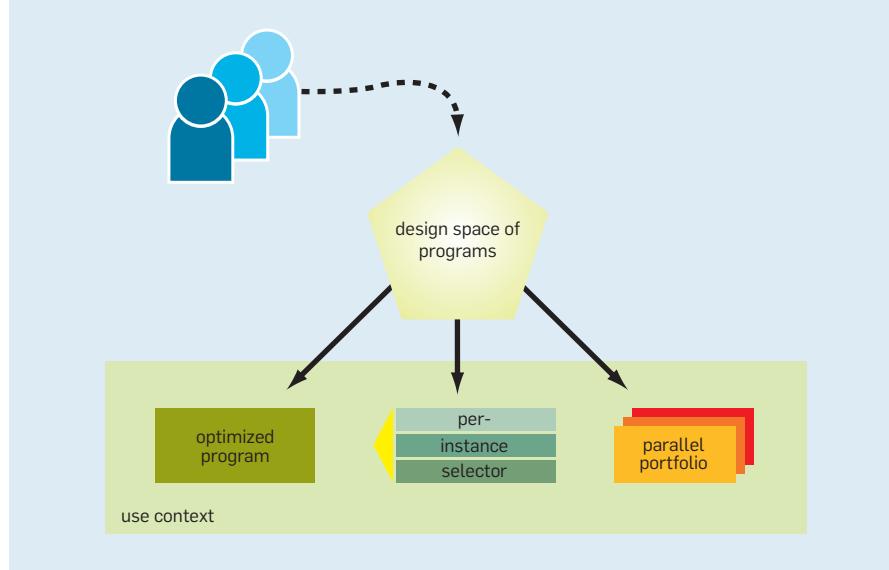
The example of SPEAR reflects the two key elements of the PbO approach:

- Specification of large, rich combinatorial design spaces of programs that solve a given problem by avoiding premature commitment to certain design choices and development of promising alternatives for parts of the design; and
- Automated generation of programs that perform well in a given use context from this specification by means of optimization techniques that realize the performance potential inherent in a given design space.

These concepts can be realized to various degrees and are present to some extent in practices already used within computing science and beyond. Our goal is to formulate and establish an approach to software development and, indeed, to the solution of computational problems that explicitly recognizes these elements, as well as provide conceptual support and tools that facilitate advanced forms of PbO.

^c These parameters control all fundamental mechanisms behind modern SAT solvers, including variable selection, clause learning, restarts, and simplification strategies.

Figure 1. Developers specify a potentially large design space of programs that accomplish a given task; from it, versions of the program optimized for different application contexts are generated automatically; per-instance selectors and parallel portfolios of programs can be derived from the same specification.



Based on observations from SPEAR (and elsewhere), we distinguish four levels of PbO:

Level 0. Settings of the parameters exposed by an existing piece of software are optimized for a given use context (characterized by, say, a set of typical input data), also known as parameter tuning or algorithm configuration;

Level 1. The design space represented by an existing piece of software is extended by exposing design choices hardwired into code. Such choices include certain magic constants, or literals that, when modified, could affect performance but not the program's correct function, as well as hidden parameters and named constants that could take different values without compromising the program's correctness. Further examples of hardwired choices are variables set to constant values but not exposed as externally accessible parameters and abandoned design alternatives, or pieces of code that could be used in addition to or instead of active code without compromising correctness but that are no longer reachable during execution of the current version of the program;

Level 2. Design choices considered during the normal course of the software-development process are actively kept and exposed to the user;

Level 3. The software-development process is structured and carried out in a way that seeks to provide design choices and alternatives in many performance-relevant components of a project; and

Level 4. The software-development process is centered on the idea of providing design choices and alternatives in all parts of a project that might benefit from them; design choices that cannot be justified convincingly are not made prematurely.

While levels 0 and 1 deal with existing software and essentially involve adding one or more phases to the development process based on PbO principles, levels 2–4 integrate PbO tightly into software creation. These higher levels of PbO typically involve assessing the cost of developing alternatives against the value ascribed to the possible gains in performance (a topic discussed later). Note that levels 3 and 4 lend themselves to a team approach, where various team members contribute alternative designs for functionally equivalent components. The first example in the table can be classified as level 0 and the second as level 1, while the late-development stages of the SPEAR SAT solver fall between levels 2 and 3.

It is possible to apply the PbO paradigm with existing methods and tools,

particularly since research on automated algorithm configuration has yielded several powerful techniques (discussed later). Nevertheless, especially at the highest levels of PbO-based software development, substantial benefit can be gained from using dedicated tools. Software development using dedicated PbO support involves three key stages (see Figure 2):

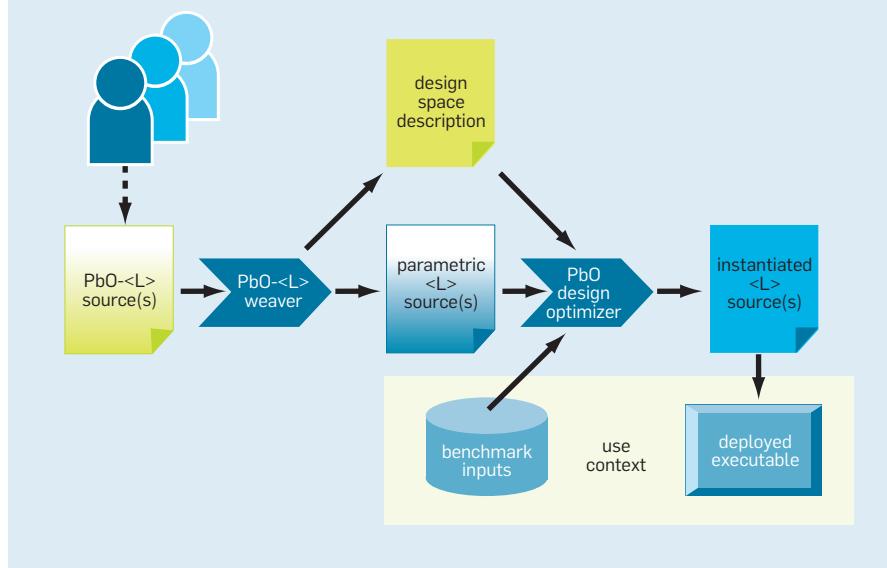
- Developers write the source code for a program in a language generically extended with constructs that explicitly declare alternative blocks of code and parameters that are to be exposed on the command line; for a programming language $\langle L \rangle$, we call the thus extended language PbO- $\langle L \rangle$. This enriched source specifies a design space of programs rather than a single program;

- A tool we call the “PbO weaver” transforms this PbO- $\langle L \rangle$ specification into a program written entirely in language $\langle L \rangle$ that exposes the design choices specified in the original source as parameters; it also produces a description of the respective design space; and

- A second tool called the “PbO design optimizer” produces from the parametric $\langle L \rangle$ -source a fully instantiated $\langle L \rangle$ -source, that is, a version of the program in which all design choices are made in a way that results in desirable performance characteristics on benchmark inputs characteristic of a given use context. This version of the program can then be deployed, in the case of compiled languages $\langle L \rangle$ (such as C and C++) after compilation.

The PbO-based approach to software development can provide substantially increased flexibility to software developers, providers, and users alike. In particular, it makes it possible to automatically customize software for optimized performance in different use contexts. This optimization can be carried out automatically by the software developer, provider, and even user; furthermore, it can, in principle, be performed at the level of a parametric executable and does not require sources to be made available to the provider or user. PbO also provides a generic way for creating software that periodically and automatically adapts itself for optimized performance as the use context changes over time. This

Figure 2. Developers produce a PbO-enhanced source code in their language of choice, $\langle L \rangle$, from which the PbO weaver generates parametric source code in pure $\langle L \rangle$, as well as a description of the design space; the PbO design optimizer uses this parametric source, along with benchmark input data, to produce a fully instantiated version of the source code, optimized specifically for deployment in the given use context.



process could be set up to take place entirely client-side or be delivered as an Internet service. In the latter case, input data from the actual application is collected client-side and transmitted to the service provider, where the PbO design optimizer is run to produce a new software configuration that is then transmitted back to the client and deployed there.

Another attractive feature of PbO involves the ability to generate multiple programs for a given purpose that are automatically combined into a per-instance selector; that is, a mechanism that selects one of them to be run on given input data based on characteristics of that data or into a parallel portfolio that runs them concurrently on the same input, optimized for a given use context. These complex designs are generated from the same design-space specification (in the form of a PbO-<L> source) that gives rise to a single, optimized program. PbO thus offers generic and automated ways of taking advantage of the fact that large design spaces typically contain programs that work well on different kinds of input data; it also offers a generic, automated way of generating parallel programs from inherently sequential sources. To construct portfolios and per-instance selectors, the PbO weaver must produce a suitably modified parametric source for the component programs, as well as for an execution controller that launches, monitors, and terminates the component programs as needed. Moreover, the PbO design optimizer must also produce fully instantiated sources (or parameter settings) for all component programs; when building a per-instance selector, it also makes use of a “feature extractor” that computes from given input data the features that serve as the basis for determining the component program to be run.

Design-Space Specification

To facilitate specification of design spaces, we introduce three basic mechanisms; the first two provide dedicated support for exposing parameters and specifying alternative blocks of code, respectively. Level 1 of PbO (explored earlier) reflects the need for both mechanisms—one required to expose magic constants and hidden param-

eters, the other to capture design alternatives that might otherwise have been abandoned. The same requirements are encountered at higher levels of PbO. The third mechanism provides lightweight support for exposing information available while a program is running; this information can be used to adaptively control design choices at runtime (by automatically generated execution controllers), as well as for debugging and empirical analysis.

All three mechanisms can, in principle, be realized within the target language used for implementing the program under development; however, this approach to design-space specification suffers from several weaknesses: First, it does not allow for easy, automatic extraction of design-space descriptions required as input for the design-optimization process. Second, it does not easily support instantiation; the process in which certain design choices made during design optimization are hardwired into the source code to produce leaner, more-efficient executables. Finally, it does not facilitate dedicated PbO support by widely used software-development environments and tools, as in, say, PbO-aware syntax highlighting, folding, and management of design alternatives. The use of conditional compilation (provided by, say, the C pre-processor) addresses lack of support for instantiation but adds substantial overhead to the meta-algorithmic optimization process (discussed later), which often involves running thousands of distinct configurations of a program.

These issues are best addressed by a generic programming-language extension providing dedicated support for exposing parameters, design alternatives, and runtime information. Such an extension would facilitate clear, explicit specification of the parts of a program representing design choices deliberately left open in a manner that is independent of the programming language used. It also provides a better basis for extending and enhancing widely used development platforms to support use of PbO.

The programming-language extension proposed here consists of four simple constructs, including two for declaring and accessing (typed) parameters, one for declaring choices, or

sets of interchangeable blocks of code representing design alternatives (possibly nested and distributed across the source code), and one for logging the current value of arbitrary (typed) expressions at runtime. Depending on PbO weaver settings, information logged this way could be written to one or more files (particularly to standard output) or to a database; it could also be sent directly to an execution controller, using other mechanisms (such as remote procedure calls and network sockets). Details on these constructs are provided in the online Appendix.

Meta-Algorithmic Optimization

Following specification of a design space, meta-algorithmic optimization procedures are used to automatically find a program with desirable performance characteristics within it. In the simplest case, these procedures determine a single, fully instantiated program with performance (measured according to a user-defined metric, such as average runtime) optimized for a given set of inputs. Since choices can be exposed as parameters (handled by the PbO weaver described in the Appendix), the problem solved by these meta-algorithmic optimization procedures corresponds to the well-known algorithm-configuration problem (sometimes called the “parameter tuning problem”), which can be described as follows: Given a target algorithm A that can be configured via a set of exposed parameters, a set of input data I and a performance metric m, find a parameter configuration of A that yields optimized performance on I, as measured by m (see, for example, Hoos,¹³ Hutter et al.,¹⁹ and Hutter et al.²⁰).

In principle, algorithm configuration can be viewed as a stochastic optimization problem (where the stochasticity stems from the performance variation observed over a set of input data or from randomization of the computation performed on the data and solved using standard stochastic-optimization procedures (see, for example, Spall³⁴). However, such procedures lack mechanisms for dealing with sets of inputs and capped runs. The issue of performance variation over input sets is important, because evaluating many

program configurations on all inputs from a given set can incur a substantial (sometimes prohibitive) computational burden that can and typically should be avoided, given that poor performance often manifests across a range of inputs. Furthermore, there are situations in which candidate configurations can be discarded without completing runs that exceed a certain time bound, considering the performance measured for other configurations; such runs can be capped, or terminated when that time bound is reached or exceeded.¹⁹

Note, too, the specification of alternative blocks of code that are central to the way a design space is constructed in PbO, necessarily leads to categorical parameters, or parameters with a discrete set of unordered values, and nested alternatives give rise to conditional parameters. Therefore, general methods for algorithm configuration used in the context of PbO must support categorical and conditional parameters, ruling out standard procedures for stochastic and numerical optimization.

Three classes of methods are specifically designed for carrying out algorithm-configuration tasks (see also Hoos¹³): Racing procedures iteratively evaluate target algorithm configurations on inputs from a given set, using statistical hypothesis tests to eliminate candidate configurations significantly outperformed by other configurations; model-free search procedures use suitably adapted search techniques, particularly stochastic local search methods (such as iterated local search) to explore potentially vast configuration spaces; and sequential model-based optimization (SMBO) methods build a response surface model that relates parameter settings to performance, using the model to iteratively identify promising parameter settings.

Racing procedures were originally introduced to solve model-selection problems in machine learning.²⁷ When adapted to the problem of selecting a program for a given task from a set of interchangeable candidates, where each candidate may correspond to a configuration of a parameterized algorithm, the key idea is to sequentially evaluate the candidates on a series

PbO lets human experts focus on the creative task of imagining possible mechanisms for solving given problems or subproblems, while the tedious job of determining what works best in a given use context is performed automatically.

of benchmark inputs and eliminate programs as soon as they fall too far behind the current incumbent, or the candidate with overall best performance at a given stage of the race. A line of work initiated by Birattari et al. in 2002 has more recently led to a procedure dubbed “Iterated F-Race” that is demonstrated effective at solving difficult algorithm-configuration tasks with up to 12 parameters⁴; there is some indication that Iterated F-Race may also be able to handle substantially more complex situations.

As of this writing, model-free search techniques, most notably the FocusedILS procedure of Hutter et al.,¹⁹ represent the state of the art in solving algorithm-configuration problems of the kind that arise in the PbO context. Along with BasicILS, another member of the ParamILS family of algorithm-configuration procedures,^{19,20} it is today the only method that supports categorical and conditional parameters, as well as capping. At the core of the ParamILS framework is Iterated Local Search, or ILS, a versatile, well-known stochastic local-search method that has been applied with great success to a range of difficult combinatorial problems (see, for example, Hoos and Stützle¹⁶). ILS iteratively performs phases of simple local search, designed to quickly reach or approach a locally optimal solution of a given problem instance, interspersed with so-called perturbation phases, to escape from local optima. Starting from a local optimum x , ILS performs one perturbation phase in each iteration, followed by a local search phase, with the aim of reaching (or approaching) a new local optimum x' . It then uses a so-called “acceptance criterion” to decide whether to continue the search process from x' or revert to the previous local optimum, x . Applying this mechanism, ILS aims to solve a given problem instance by exploring the space of its locally optimal solutions. ParamILS performs iterated local search in the configuration space of a given parametric algorithm.

While BasicILS, the simplest variant of ParamILS, evaluates candidate configurations based on a fixed number of target algorithm runs, the more sophisticated FocusedILS procedure uses a heuristic mechanism to per-

form runs on a candidate configuration only as long as that configuration appears promising compared to the current incumbent; it therefore avoids wasting computational effort on configurations easily recognized as performing poorly. The use of FocusedILS for configuring highly parametric software has led to substantial improvement in the state of the art in solving prominent classes of SAT,^{18,24} mixed-integer programming,²¹ and planning problems.³⁵ In the case of mixed-integer programming, our group used FocusedILS to configure 76 parameters of the widely used commercial CPLEX software (searching within a design space of $1.9 \cdot 10^{47}$ configurations of the solver), resulting in up to 50-fold speedups over the extensively optimized default configuration.²¹

SMBO procedures for algorithm configuration are based on the idea of using the information gained from parameter configurations evaluated so far to build and maintain a response surface model that directly captures the dependence of target-algorithm performance on parameter settings; this model is used to determine promising configurations at any stage of an iterative model-based search procedure. Almost all existing work on sequential model-based optimization focuses on a setting known as “black-box function optimization,” with resulting procedures suffering from the same shortcomings as standard numerical optimization procedures, in that they do not support categorical and conditional parameters nor provide mechanisms for effectively dealing with sets of inputs and capping of runs. However, two of these limitations were overcome in 2011 by a procedure dubbed “Sequential Model-based Algorithm Configuration,” or SMAC, handling categorical parameters while exploiting the fact that performance is evaluated on a set of inputs.²² Evidence suggests that SMAC can, at least on some challenging configuration benchmarks, reach and sometimes exceed the performance of FocusedILS. We expect further work will lead to SMBO-based procedures that turn out to be useful for solving design-optimization tasks in the context of PbO, particularly when the parameter response of

a given target algorithm is reasonably regular and performance evaluations are very costly.

The idea of automatically constructing a per-instance algorithm selector from a single parametric design was recently explored by Xu et al.³⁹ and, independently, by Kadioglu et al.²³ In each, a given feature extractor was used to compute a vector of features from the given input to be processed. The method by Kadioglu et al., dubbed ISAC, uses a combination of clustering based on the feature values and automatic algorithm configuration to produce an algorithm selector. The Hydra procedure by Xu et al. iteratively adds configurations of a given program to the set available to the per-instance selector; in each iteration, Hydra automatically determines an additional configuration to maximally improve the performance obtained when building a selector using the thus extended set of configurations. In principle, both Hydra and ISAC can make use of arbitrary feature extractors and algorithm configuration procedures. Whereas ISAC produces a single selector based on a number of components automatically determined by the G-means algorithm,¹² Hydra builds a series of selectors; the longer it runs, the more configurations are available for selection, and the better the expected performance of the resulting selector. Furthermore, Hydra makes use of arbitrary selector builders; Xu et al.³⁹ used a procedure based on the regression-based performance predictor underlying the well-known SATzilla approach,^{30,40} though many alternatives are possible.

Procedures for building per-instance algorithm selectors fit naturally into the context of PbO. To construct per-instance selectors from a given design-space specification, the PbO design optimizer uses a procedure (such as Hydra or ISAC) to obtain a suitable set of optimized programs. The design optimizer would also produce an execution manager that first calls the feature extractor provided by the user (specific to the given computational tasks but not dependent on the design of the program used to achieve it), then selects the component algorithm to be run based on the resulting input features.

Although algorithm portfolios have been investigated (see, for example, Gomes and Selman¹⁰ or Huberman et al.¹⁷), to the best of our knowledge effective methods for automatically constructing portfolios from a single parametric design have not yet been developed, though we expect such methods to be available soon. The PbO design optimizer can then use a portfolio-construction procedure to obtain a set of programs from a given design-space specification and produce an execution manager to coordinate their execution. In the simplest case, where the component solvers run independently in parallel on the same input data, the execution manager starts each component program and processes the results from these runs when the component programs terminate. When applied to a task in which candidate programs differ only in the time they require for processing a given input (as in sorting or solving SAT), as soon as the first component program terminates successfully, the execution manager aborts all remaining component programs and returns the result from the one successful run. In the case of programs designed to solve optimization tasks, the execution manager monitors the best solutions produced by each component solver, providing the best of them at any given time.

Cost and Concerns

Readers might raise several concerns regarding PbO-based software development. The first pertains to the cost incurred by the approach in terms of computational resources and human development effort. It might seem that, due to its use of compute-intensive meta-algorithmic optimization procedures, PbO always requires large amounts of computational resources. However, while meta-algorithmic optimization in a large design space can be computationally expensive, it has been shown to yield good results at relatively modest computational cost in many cases.^{18,21,35} Furthermore, the meta-algorithmic optimization techniques mentioned earlier tend to produce increasingly better results as they are run for longer and longer times; moreover, they can all be adapted to make use of parallel computation to

more effectively search potentially very large design spaces.¹⁴

Since PbO aims to replace human development effort with computation, additional human effort incurred by the approach is of special concern. Level 0, the most basic form of PbO, causes no such overhead, and the added effort at level 1 can be minimized through effective, lightweight mechanisms for exposing design choices.^d Starting at level 2, conceiving, implementing, and testing design alternatives requires additional human effort. This development cost must be outweighed by the gains in performance a software developer might reasonably hope to achieve by optimizing performance-critical parts of a design. Using this criterion, even at levels 3 and 4, the development of design choices and alternatives may well focus on a relatively small number of key components of a complex software system. At the same time, in cases where performance matters sufficiently, the overhead associated with higher levels of PbO is at least partially offset by the substantial human effort otherwise expended for manual exploration of design choices.

Rather disturbingly, it might seem that when dealing with the large, combinatorial spaces of programs key to the PbO paradigm, the occurrence of bugs would be amplified to the point where testing and debugging becomes a major burden, if not completely infeasible. However, because design alternatives for individual mechanisms and components can be tested separately, the combinatorial set of programs to be checked is effectively reduced to a set that grows linearly with the number of choices and design alternatives available at each choice point. While there is potential for error conditions arising only in particular instantiations of multiple design choices, such conditions are mostly avoided by following sound practices regarding encapsulation of program code and

data structures, along with appropriate use of unit testing. Furthermore, as observed in 2010 by Hutter et al.,²¹ design-optimization tools (such as “automated algorithm configurators”) make it possible to find previously unknown bugs in widely used software developed through traditional methods; we expect the same to hold for PbO-based software development.

A final concern follows from the observation that software optimization for a narrowly defined use context can lead to brittle performance. It is prominent in machine learning, which offers various techniques for addressing it. A combination of judicious practices for constructing input datasets in the optimization process, appropriately defined optimization objectives, and suitable methods for assessing the performance of candidate designs appears to be effective in avoiding brittle performance and poor generalization beyond narrowly defined classes of input data.

Robust performance is traditionally important in situations where important features of the input data to be processed might change over time. The PbO paradigm offers an attractive way to deal with such situations based on the idea of automatically adapting the program design, so, at any given time, a program is well suited for the current input data, an idea closely related to the concept of lifelong learning. This adaptation can be achieved by using the PbO design optimizer to automatically generate new programs optimized for input data representative of the current use context. This process takes place after the initial design phase, in the actual application context, and does not involve human designers. Moreover, it can be carried out by a deployed system involving a highly parametric program, a meta-algorithmic optimization procedure, and a mechanism for deciding which input data encountered over the system’s lifetime is to be used when assessing the performance of candidate program designs.

Related Work

Efforts conceptually related to the ideas behind PbO can be traced back more than 30 years to the work of Rice³² in the mid-1970s, but the powerful optimization and machine-learning tech-

niques, as well as the computational environments required to carry out the automated design optimization at the heart of PbO, have only recently been readily available.

PbO is a logical extension of existing work on parameter tuning (see, for example, Adenso-Diaz and Laguna¹ or Birattari et al.⁴) automated algorithm configuration (see, for example, Hutter et al.¹⁸ or KhudaBukhsh et al.²⁴) and automated algorithm selection (see, for example, Guerri and Milano,¹¹ Leyton-Brown et al.,²⁵ or Xu et al.⁴⁰), as well as of the more general approach of computer-aided algorithm design¹⁵; it is also complementary to work on algorithm portfolios (see, for example, Gagliolo,⁹ Gomes and Selman,¹⁰ or Huberman et al.¹⁷) and self-adaptation (see, for example, Battiti et al.,³ Carchrae and Beck,⁵ or Da Cost et al.⁷), which can benefit from PbO and be leveraged in PbO-based software development. We also see connections with work in algorithm synthesis (see, for example, Monette et al.²⁸ or Westfold and Smith³⁷), algorithm engineering (see, for example, Sanders and Schultes³³), and meta-learning (see, for example, Vilalta and Drissi³⁶).

Furthermore, many studies clearly exhibit key elements of the PbO approach and bear witness to its benefits, including our own work on algorithm configuration for complete and incomplete SAT solvers,^{18,24} for several well-known solvers for mixed-integer programming problems,²¹ and for two well-known general-purpose planning systems.³⁵ While these studies focused on optimizing the performance of software for solving NP-hard problems, a broad range of similar work involves software running in polynomial time.^e For example, Whaley et al.³⁸ automatically performed mostly low-level optimizations of performance-critical, basic linear algebra routines used in numerous applications; Pan and Eigemann³¹ automatically determined performance-maximizing combinations of compiler optimizations for a given program or program section; Diao et al.⁸ automatically configured a database server for minimal response

^d Experience shows that even the rather modest effort required to expose an additional parameter when using languages like C and C++ can discourage developers, just like the overhead of frequent recompilation severely limits their exploration of design choices accessed through conditional compilation or source-level modifications.

^e Optimization of this software, on the other hand, is a combinatorial problem with high computational complexity.

time in an e-commerce application; and Li et al.²⁶ automatically created hybrid sorting algorithms that outperform those provided by several widely used libraries, including the C++ Standard Template Library.

Where the Road Goes...

The PbO paradigm offers numerous benefits to software developers and users alike, including better performance of programs created this way and easier, more effective adaptation to different (and changing) use contexts, as well as better use of human capabilities and skills throughout the development process.

To be effective, PbO needs to be used in combination with other techniques and established practices. In particular, careful consideration of design patterns, memory-access and communication patterns, data organization, and threading will still be crucially important for achieving high performance in many cases, as will performance-profiling approaches. PbO should be seen as complementing, rather than superseding, these considerations, which conversely inform and constrain the design choices realized in the context of a PbO-based development process. The cost of PbO induces additional constraints and may in certain cases limit the degree to which the approach can be applied. Still, many areas of computing science and its applications have much to gain from PbO, particularly for software using techniques from artificial intelligence, machine learning, and data mining, as well as simulation software, performance-critical procedures from standard libraries, and even data transmissions protocols, basically any situation involving heuristic design choices.

While lower levels of PbO, in combination with existing tools, are already able to achieve substantial benefits, we believe the full potential of PbO is realized through higher levels of PbO-based software development and dedicated support in the form of the language extensions and tools outlined here. A first version of a weaver for PbO-C was implemented by the author and now available at <http://www.prog-by-opt.net>. PbO design optimization can be achieved

Because it enables empirical investigation into the interaction between problem-instance characteristics and the efficacy of certain solver components, PbO promises to facilitate insight into what renders certain problems so difficult to solve.

through readily available automated algorithm-configuration procedures (such as ParamILS^{19,20}), and we expect even better performing procedures to be available within the next two years. Similarly, meta-algorithmic procedures that effectively produce per-instance algorithm selectors from a single, highly parametric design are available today (see, for example, Xu et al.³⁹) and will likely be further improved in the near future. We are currently working on automated procedures for generating parallel portfolios from a given design-space specification and expect to obtain useful results soon. We plan to integrate these (and possibly other) meta-algorithmic optimization procedures into a single PbO design optimizer that facilitates their use in the context of PbO-based software development.

The High Performance Algorithm Laboratory (HAL) environment²⁹ is designed to support computer-aided design and empirical analysis of high-performance algorithms through ready-to-use, state-of-the-art analysis and design procedures. HAL provides an ideal platform for realizing and operating an integrated PbO design optimizer and could thus provide strong support for PbO-based software development. Furthermore, extensions and enhancements of widely used development platforms, particularly the Eclipse integrated development environment (<http://www.eclipse.org>), will provide useful support for PbO-based software development. Besides syntax highlighting and folding for PbO constructs, we envision tools that support developers tracking and navigating parameters and choices (especially distributed choices) declared in PbO sources, and in using PbO weavers and optimizers in their various modes.

We expect PbO to also facilitate scientific insight into the efficacy of algorithms and their components, as well as into the empirical complexity of computational problems. For example, to measure the extent to which a particular instance of a design choice contributes to overall performance, one would simply remove that instance (or instruct the weaver to ignore it) and compare the performance obtained in one or more use

contexts when optimizing within this reduced design space with the performance obtained from the original design space. Further analysis of the differences between the two designs could then produce insight into the degree to which other design choices might compensate for the effects of eliminating that choice instance from the design space. Comparing designs optimized for different use contexts can reveal interactions between characteristics of a program's inputs and the mechanisms that should be used to achieve good performance on those inputs. Finally, because it enables empirical investigation into the interaction between problem instance characteristics and the efficacy of certain solver components, PbO promises to facilitate insight into what renders certain problems so difficult to solve.

While much work remains to realize the full potential of the approach, PbO will change the way developers and users create, use, and study software. While PbO will be especially effective in the context of solving NP-hard problems, where general insight into practically effective solution methods is limited, we are convinced it will prove useful on a much broader scale.

Acknowledgments

Some of the ideas discussed here have their roots in joint work and discussions with Frank Hutter, Chris Fawcett, Kevin Leyton-Brown, and Catherine Yelick. Much of the work on automated algorithm selection and configuration and parameter tuning was carried out by my research group at the University of British Columbia, primarily involving Frank Hutter, Lin Xu, Kevin Leyton-Brown, and Kevin Murphy, as well as Thomas Stützle at the Université Libre de Bruxelles, to whom I am grateful for fruitful and ongoing collaboration. I also gratefully acknowledge helpful comments by Sam Bayless and the anonymous reviewers on earlier drafts of this work. 

References

1. Adenso-Díaz, B. and Laguna, M. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research* 54, 1 (Jan.–Feb. 2006), 99–114.
2. Babić, D. and Hu, A.J. Structural abstraction of software verification conditions. In *Proceedings of the 19th International Computer Aided Verification Conference*, Vol. 4590 LNCS. Springer-Verlag, Berlin/Heidelberg, 2007, 366–378.
3. Battiti, R., Brunato, M., and Mascia, F. *Reactive Search and Intelligent Optimization*. Operations Research/Computer Science Interfaces Series, Vol. 45. Springer, 2008.
4. Birattari, M., Yuan, Z., Balaprakash, P., and Stützle, T. F-Race and Iterated F-Race: An overview. In *Experimental Methods for the Analysis of Optimization Algorithms*. Springer-Verlag, Berlin/Heidelberg, 2010, 311–336.
5. Carchrae, T. and Beck, J. Applying machine learning to low knowledge control of optimization algorithms. *Computational Intelligence* 21, 4 (Nov. 2005), 373–387.
6. Cheeseman, P., Kanefsky, B., and Taylor, W.M. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1991, 331–337.
7. Da Costa, L., Fialho, Á., Schoenauer, M., and Sebag, M. Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*. ACM Press, New York, 2008, 913–920.
8. Diao, Y., Eskesen, F., Froehlich, S., Hellerstein, J.L., Spainhower, L., and Surendra, M. Generic online optimization of multiple configuration parameters with application to a database server. In *Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, Vol. 2867 LNCS. Springer-Verlag, Berlin/Heidelberg, 2003, 3–15.
9. Gagliolo, M. and Schmidhuber, J. Dynamic algorithm portfolios. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*; <http://anytime.cs.umass.edu/aimath06/>
10. Gomes, C.P. and Selman, B. Algorithm portfolios. *Artificial Intelligence* 126, 1–2 (Feb. 2001), 43–62.
11. Guerrini, A. and Milano, M. Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the 16th European Conference on Artificial Intelligence*. IOS Press, Amsterdam, 2004, 475–479.
12. Hamerly, G. and Elkan, C. Learning the K in k-means. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. MIT Press, Cambridge, MA, 2004, 281–288.
13. Hoos, H.H. Automated algorithm configuration and parameter tuning. In *Autonomous Search*, Y. Hamadi and F. Saubion, Eds. Springer-Verlag, 2011.
14. Hoos, H. *Programming by Optimisation*. Technical Report TR 2010-14. Department of Computer Science, University of British Columbia, Vancouver, 2010.
15. Hoos, H. *Computer-Aided Design of High-Performance Algorithms*. Technical Report TR 2008-16. Department of Computer Science, University of British Columbia, Vancouver, 2008.
16. Hoos, H.H. and Stützle, T. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann Publishers, San Francisco, 2004.
17. Huberman, B., Lukose, R., and Hogg, T. An economics approach to hard computational problems. *Science* 275, 5296 (Jan. 1997), 51–54.
18. Hutter, F., Babić, D., Hoos, H.H., and Hu, A.J. Boosting verification by automatic tuning of decision procedures. In *Proceedings of Formal Methods in Computer-Aided Design*. IEEE Computer Society Press, Los Alamitos, CA, 2007, 27–34.
19. Hutter, F., Hoos, H., Leyton-Brown, K., and Stützle, T. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36 (Sept.-Dec. 2009), 267–306.
20. Hutter, F., Hoos, H., and Stützle, T. Automatic algorithm configuration based on local search. In *Proceedings of the 22nd National Conference on Artificial Intelligence*. AAAI Press, Palo Alto, CA, 2007, 1152–1157.
21. Hutter, F., Hoos, H.H., and Leyton-Brown, K. Automated configuration of mixed integer programming solvers. In *Proceedings of the Seventh International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Vol. 6140 LNCS. Springer-Verlag, Berlin/Heidelberg, 2010, 186–202.
22. Hutter, F., Hoos, H.H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization*, Vol. 6683 LNCS. Springer-Verlag, Berlin/Heidelberg, 2011, 507–523.
23. Kadioglu, S., Malitsky, Y., Sellmann, M., and Tierney, K. ISAC: An instance-specific algorithm configuration. In *Proceedings of the 19th European Conference on Artificial Intelligence*. IOS Press, Amsterdam, 2010, 751–756.
24. KhudaBukhsh, A., Xu, L., Hoos, H., and Leyton-Brown, K. SATenstein: Automatically building local search SAT solvers from components. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. AAAI Press, Palo Alto, CA, 2009, 517–524.
25. Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., and Shoham, Y. A portfolio approach to algorithm selection. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, San Francisco, 1542–1543.
26. Li, X., Garzani, M.J., and Padua, D. Optimizing sorting with genetic algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society Press, Washington, D.C., 2005, 99–110.
27. Maron, O. and Moore, A.W. Hoeffding races: Accelerating model selection search for classification and function approximation. In *Proceedings of the Seventh Conference on Advances in Neural Information Processing Systems*. Morgan Kaufmann Publishers, San Francisco, 1994, 59–66.
28. Monette, J.N., Deville, Y., and Hentenryck, P.V. Aeon: Synthesizing scheduling algorithms from high-level models. *Operations Research and Cyber-Infrastructure Series*, Vol. 47. Springer Science+Business, New York, 2009, 43–59.
29. Nell, C.W., Fawcett, C., Hoos, H.H., and Leyton-Brown, K. HAL: A framework for the automated design and analysis of high-performance algorithms. In *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization*, Vol. 6683 LNCS. Springer-Verlag, Berlin/Heidelberg, 2011, 600–615.
30. Nudelman, E., Leyton-Brown, K., Devkar, A., Shoham, Y., and Hoos, H.H. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Principles and Practice of Constraint Programming*, Vol. 3258 LNCS. Springer-Verlag, Berlin/Heidelberg, 2004, 438–452.
31. Pan, Z. and Eigenmann, R. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society Press, Washington, D.C., 2006, 319–332.
32. Rice, J.R. The algorithm selection problem. *Advances in Computers* 15 (1976), 65–118.
33. Sanders, P. and Schultes, D. Engineering fast route planning algorithms. In *Proceedings of the Sixth International Workshop on Experimental Algorithms*, Vol. 4525 LNCS. Springer-Verlag, Berlin/Heidelberg, 2007, 23–36.
34. Spall, J. *Introduction to Stochastic Search and Optimization*. John Wiley & Sons, Inc., New York, 2003.
35. Vallati, M., Fawcett, C., Gerevini, A., Hoos, H.H., and Saetti, A. Automatic generation of efficient domain-optimized planners from generic parametrized planners. In *Proceedings of the Eighth RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 2011; <http://ijcai-11.iiia.csic.es/files/proceedings/RCRA2011-proceedings.pdf>
36. Vallta, R. and Drissi, Y. A perspective view and survey of meta-learning. *Artificial Intelligence Review* 18, 2 (Oct. 2002), 77–95.
37. Westfold, S.J. and Smith, D.R. Synthesis of efficient constraint-satisfaction programs. *Knowledge Engineering Review* 16, 1 (Mar. 2001), 69–84.
38. Whaley, R.C., Petitet, A., and Dongarra, J.J. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* 27, 1–2 (Jan. 2001), 3–35.
39. Xu, L., Hoos, H., and Leyton-Brown, K. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*. AAAI Press, Palo Alto, CA, 2010, 210–216.
40. Xu, L., Hutter, F., Hoos, H.H., and Leyton-Brown, K. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research* 32 (May-Aug. 2008), 565–606.

Holger H. Hoos (hoos@cs.ubc.ca) is a professor in the computer science department of the University of British Columbia, Vancouver, B.C., and a faculty associate in the Peter Wall Institute for Advanced Studies of the University of British Columbia, Vancouver, B.C.