Programming by Optimisation:

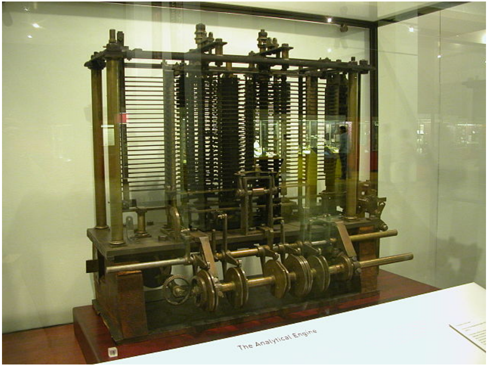
Towards a new Paradigm for Developing High-Performance Software

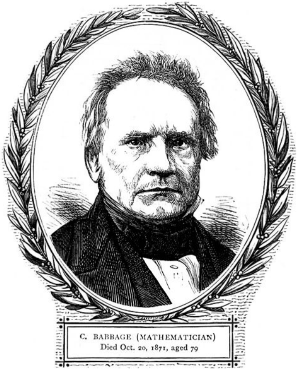
Holger H. Hoos

BETA Lab
Department of Computer Science
University of British Columbia
Canada

ICAPS 2013
Rome, Italy, 2013/06/11

The age of machines





"As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise – by what course of calculation can these results be arrived at by the machine in the shortest time?"

(Charles Babbage, 1864)

BBC Mobile

News | Sport | Weather | Travel | TV | Radio

NEWS TECHNOLOGY

Home | US & Canada | Latin America | UK | Africa | Asia | Europe | Mid-East | Business | Health | Sci/Environment | Tech

22 August 2011 Last updated at 20:42 ET

1.4K Share

When algorithms control the world

By Jane Wakefield
Technology reporter

If you were expecting some kind of warning when computers finally get smarter than us, then think again.

There will be no soothing HAL 9000-type voice informing us that our human services are now surplus to requirements.

In reality, our electronic overlords are already taking control, and they are doing it in a far more subtle way than science fiction would have us believe.

Their weapon of choice - the algorithm.

Behind every smart web service is some even smarter web code. From the web retailers - calculating what books and films we might be interested in, to Facebook's friend finding and image tagging services, to the search engines that guide us around the net.

Algorithms are spreading their influence around the globe

It is these invisible computations that increasingly control how we interact with our electronic world.

At last month's TEDGlobal conference, algorithm expert Kevin Slavin delivered one of the tech show's most "sit up and take notice" speeches where he warned that the "maths that computers use to decide stuff" was infiltrating every aspect of our lives.

Related Stories

Are search engines skewing objectivity?

Robot reads minds to train itself

The age of computation


BBC Mobile News | Sport | Weather | Travel | TV | Radio

NEWS TECHNOLOGY

Home | US & Canada | Latin America | UK | Africa | Asia | Europe | Mid-East | Business | Health | Sci/Environment | Tech

22 August 2011 Last updated at 20:42 ET

1.4K Share

When algorithms control the world

By Jane Wakefield
Technology reporter

If you were expecting some kind of warning when computers finally get smarter than us, then think again.

There will be no soothing HAL 9000-type voice informing us that our human services are now surplus to requirements.

In reality, our electronic overlords are already taking control, and they are doing it in a far more subtle way than science fiction would have us believe.

Their weapon of choice - the algorithm.

Behind every smart web service is some even smarter web code. From the web retailers - calculating what books and films we might be interested in, to Facebook's friend finding and image tagging services, to the search engines that guide us around the net.

Algorithms are spreading their influence around the globe

It is these invisible computations that increasingly control how we interact with our electronic world.

At last month's TEDGlobal conference, algorithm expert Kevin Slavin delivered one of the tech show's most "sit up and take notice" speeches where he warned that the "maths that computers use to decide stuff" was infiltrating every aspect of our lives.

Related Stories

Are search engines skewing objectivity?

Robot reads minds to train itself

"The maths[!] that computers use to decide stuff [is] infiltrating every aspect of our lives."

- financial markets
- social interactions
- cultural preferences
- artistic production
- ...

Performance matters ...

- computation speed (time is money!)
- energy consumption (battery life, ...)
- quality of results (cost, profit, weight, ...)

... increasingly:

- globalised markets
- just-in-time production & services
- tighter resource constraints

Example: Resource allocation

- resources $>$ demands $\rightsquigarrow$ many solutions, easy to find
  economically wasteful
  $\rightsquigarrow$ reduction of resources / increase of demand
- resources $<$ demands $\rightsquigarrow$ no solution, easy to demonstrate
  lost market opportunity, strain within organisation
  $\rightsquigarrow$ increase of resources / reduction of demand
- resources $\approx$ demands
  $\rightsquigarrow$ difficult to find solution / show infeasibility

This tutorial:

new approach to software development, leveraging ...

- human creativity
- optimisation & machine learning
- large amounts of computation / data

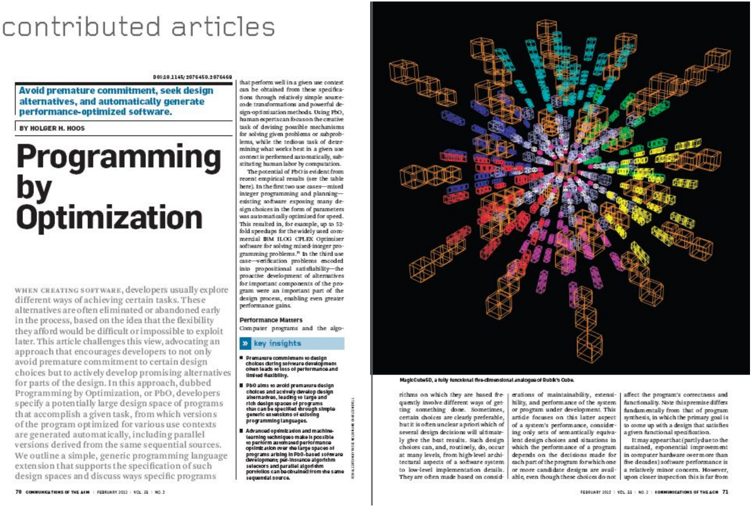
Key idea:

- program $\rightsquigarrow$ (large) space of programs
- encourage software developers to
  - avoid premature commitment to design choices
  - seek & maintain design alternatives
- automatically find performance-optimising designs for given use context(s)

$\Rightarrow$ Programming by Optimisation (PbO)

Outline

1. Introduction
2. Vision & promise of PbO
3. Design space specification
4. Design optimisation
5. Cost & concerns
6. The road ahead – towards main-stream use of PbO



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 algorithms 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 considerations 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

» 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.

Communications of the ACM, 55(2), pp. 70–80, February 2012

www.prog-by-opt.net

Example: SAT-based software verification

Hutter, Babić, HH, Hu (2007)

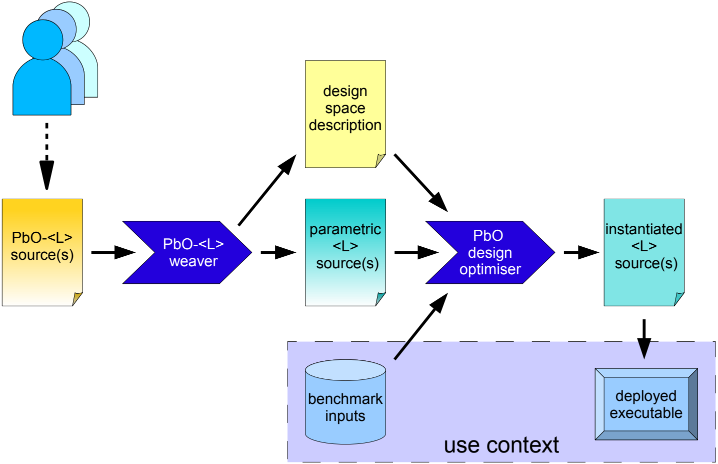
- **Goal:** Solve SAT-encoded software verification problems as fast as possible
- new DPLL-style SAT solver SPEAR (by Domagoj Babić)
  = highly parameterised heuristic algorithm
  (26 parameters, $\approx 8.3 \times 10^{17}$ configurations)
- manual configuration by algorithm designer
- automated configuration using ParamILS, a generic algorithm configuration procedure
  Hutter, HH, Stützle (2007)

SPEAR: Performance on software verification benchmarks

solver	num. solved	mean run-time
MiniSAT 2.0	302/302	161.3 CPU sec
SPEAR original	298/302	787.1 CPU sec
SPEAR generic. opt. config.	302/302	35.9 CPU sec
SPEAR specific. opt. config.	302/302	1.5 CPU sec

- $\approx$ 500-fold speedup through use automated algorithm configuration procedure (ParamILS)
- new state of the art
  (winner of 2007 SMT Competition, QF_BV category)

Software development in the PbO paradigm



Levels of PbO:

Level 4: Make no design choice prematurely that cannot be justified compellingly.

Level 3: Strive to provide design choices and alternatives.

Level 2: Keep and expose design choices considered during software development.

Level 1: Expose design choices hardwired into existing code (magic constants, hidden parameters, abandoned design alternatives).

Level 0: Optimise settings of parameters exposed by existing software.

Success in optimising speed:

Application, Design choices	Speedup	PbO level
SAT-based software verification (SPEAR), 41 Hutter, Babić, HH, Hu (2007)	4.5–500 ×	2–3
AI Planning (LPG), 62 Vallati, Fawcett, Gerevini, HH, Saetti (2011)	3–118 ×	1
Mixed integer programming (CPLEX), 76 Hutter, HH, Leyton-Brown (2010)	2–52 ×	0

... and solution quality:

University timetabling, 18 design choices, PbO level 2–3
$\rightsquigarrow$ new state of the art; UBC exam scheduling
Fawcett, Chiarandini, HH (2009)

Machine learning / Classification, 786 design choices, PbO level 0–1
$\rightsquigarrow$ outperforms specialised model selection & hyper-parameter optimisation methods from machine learning
Thornton, Hutter, HH, Leyton-Brown (2012–13)

Mixed Integer Programming (MIP)

Hutter, HH, Leyton-Brown, Stützle (2009); Hutter, HH, Leyton-Brown (2010)

- MIP is widely used for modelling optimisation problems
- MIP solvers play an important role for solving broad range of real-world problems

CPLEX:

- prominent and widely used commercial MIP solver
- exact solver, based on sophisticated branch & cut algorithm and numerous heuristics
- 159 parameters, 81 directly control search process

"A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models."

[CPLEX 12.1 user manual, p. 478]

Automatically Configuring CPLEX:

- starting point: factory default settings
- 63 parameters (some with 'AUTO' settings)
- 1.38×10^{37} configurations
- configurator: FocusedILS 2.3 (Hutter *et al.* 2009)
- performance objective: minimal mean run-time
- configuration time: 10×2 CPU days

CPLEX on various MIPS benchmarks

Benchmark	Default performance [CPU sec]	Optimised performance [CPU sec]	Speedup factor
BCOL/CONIC.SCH	5.37	2.35 (2.4 ± 0.29)	2.2
BCOL/CLS	712	23.4 (327 ± 860)	30.4
BCOL/MIK	64.8	1.19 (301 ± 948)	54.4
CATS/REGIONS200	72	10.5 (11.4 ± 0.9)	6.8
RNA-QP	969	525 (827 ± 306)	1.8

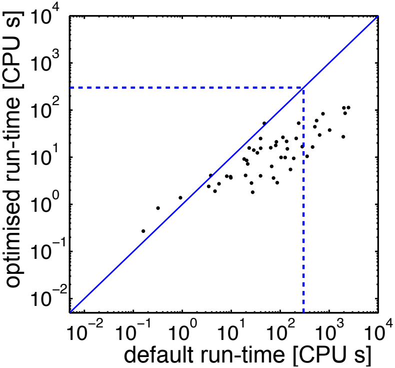
(Timed-out runs are counted as $10 \times$ cutoff time.)

CPLEX on various MIPS benchmarks

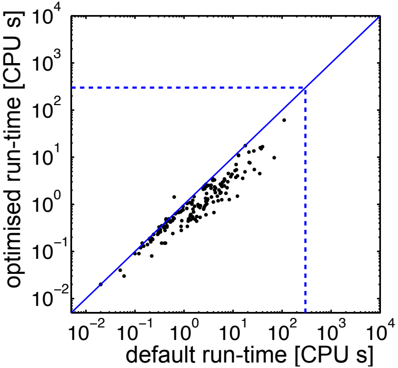
Benchmark	Default performance [CPU sec]	Optimised performance [CPU sec]	Speedup factor
BCOL/CONIC.SCH	5.37	2.35 (2.4 ± 0.29)	2.2
BCOL/CLS	712	23.4 (327 ± 860)	30.4
BCOL/MIK	64.8	1.19 (301 ± 948)	54.4
CATS/REGIONS200	72	10.5 (11.4 ± 0.9)	6.8
RNA-QP	969	525 (827 ± 306)	1.8

(Timed-out runs are counted as $10 \times$ cutoff time.)

CPLEX on BCOL/CLS



CPLEX on BCOL/CONIC.SCH



Planning

Vallati, Fawcett, HH, Gerevini, Saetti (2011)

- classical, well-studied AI challenge
- many variations, domains (explicitly specified)

LPG:

- state-of-the-art, versatile system for plan generation, plan repair and incremental planning for PDDL2.2 domains
- based on stochastic local search over partial plans
- 62 parameters, over 6.5×10^{17} configurations
  4 of these previously "magic constants",
  50 hidden (= undocumented)
- automated configuration using FocusedILS 2.3

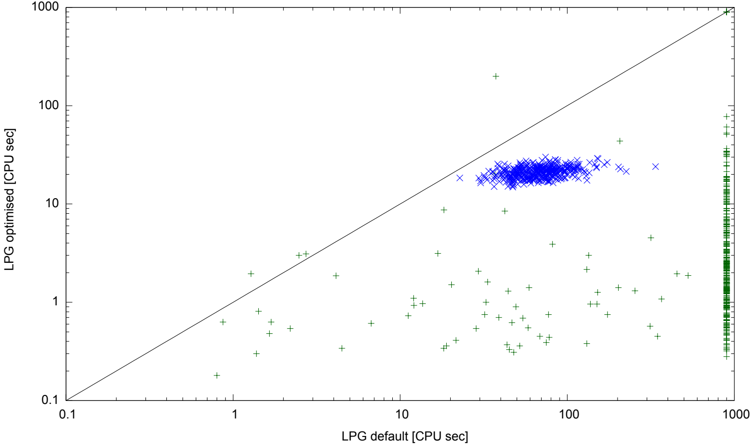
$\rightsquigarrow$ ParLPG

LPG on various planning domains

Domain	Default performance [CPU sec] (% solved)	Optimised performance [CPU sec] (% solved)
`Blocksworld`	105.3 (98.8%)	4.29 (100%)
`Depots`	78.1 (90.3%)	5.7 (98.5%)
`Gold-miner`	94.4 (90.5%)	1.6 (100%)
`Matching-BW`	93.8 (15.8%)	5.6 (97.8%)
`N-Puzzle`	321 (85%)	31.2 (86.8%)
`Rovers`	72.2 (100%)	21.2 (100%)
`Satellite`	64 (100%)	1.3 (100%)
`Sokoban`	24.6 (75.8%)	1.19 (96.5%)
`Zenotravel`	103.7 (100%)	11.1 (100%)

Run-time cutoff for evaluation: 600 CPU sec

LPG on Matching-BW, Rovers (hard instances)



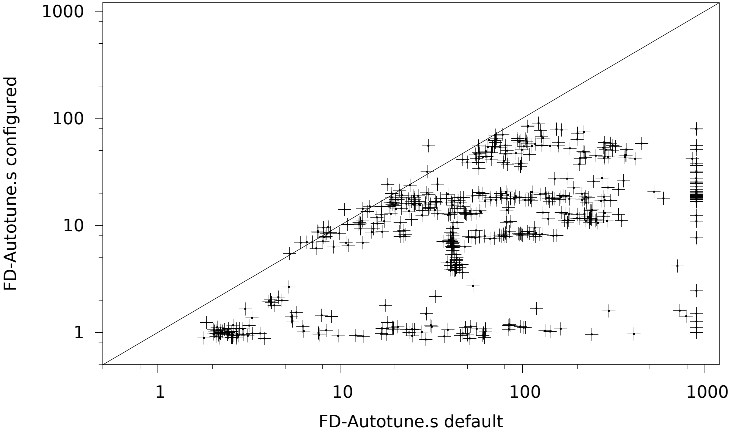
(domain-specific configurations; run-time cutoff for evaluation: 900 CPU sec)

Configuring Fast Downward: FD-Autotune

Fawcett, Helmert, HH, Karpas, Röger, Seipp (2011)

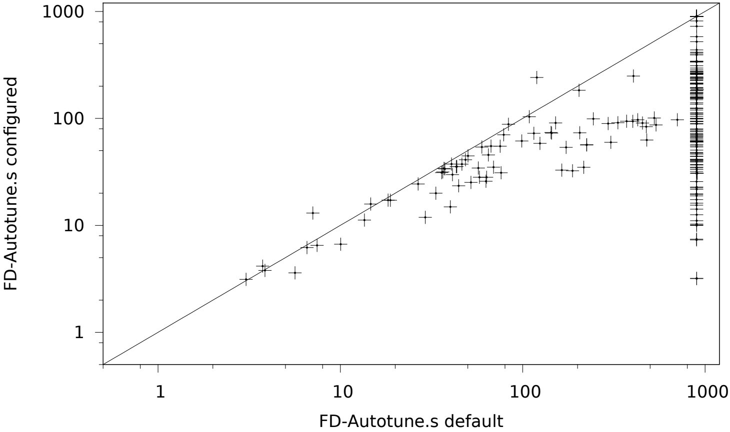
- used new, highly parameterised IPC-2011 version of Fast Downward
- design space includes combinations of heuristics, chaining of search procedures
- 45 parameters, 2.99×10^{13} configurations
- configured using FocusedILS, 10 runs of 5 CPU days each per domain
- objective: minimum running time for finding satisficing plan

FD-Autotune on IPC-2011 domains (training instances)



(domain-specific configurations)

FD-Autotune on IPC-2011 domains
(test instances)



(domain-specific configurations)

IPC 2011 Learning Track – Success!

- Separate submissions for ParLPG, FD-Autotune
- Integrated systems realised using HAL experimentation system (Nell, Fawcett, HH, Leyton-Brown 2011)
- FD-Autotune: 2nd place
- ParLPG: contributed substantially to performance of winner, PbP2 (Gerevini, Saetti, Vallati 2009)

Automated Selection and Hyper-Parameter Optimization of Classification Algorithms

Thornton, Hutter, HH, Leyton-Brown (2012–13)

Fundamental problem:

Which of many available algorithms (models) applicable to given machine learning problem to use, and with which hyper-parameter settings?

Example: WEKA contains 39 classification algorithms,
3×8 feature selection methods

Our solution, Auto-WEKA

- select between the $39 \times 3 \times 8$ algorithms using high-level categorical choices
- consider hyper-parameters for each algorithm
- solve resulting algorithm configuration problem using general-purpose configurator SMAC
- first time joint algorithm/model selection + hyperparameter-optimisation problem is solved

Automated configuration process:

- configurator: SMAC
- performance objective: cross-validated mean error rate
- time budget: $4 \times 30 CPUhours$

Selected results (mean error rate)

					Auto-WEKA	
Dataset	#Instances	#Features	#Classes	Best Def.	TPE	SMAC
Semeion	1115+478	256	10	8.18	8.26	**5.08**
KR-vs-KP	2237+959	37	2	0.31	0.54	**0.31**
Waveform	3500+1500	40	3	14.40	**14.23**	14.42
Gisette	4900+2100	5000	2	2.81	3.94	**2.24**
MNIST Basic	12k+50k	784	10	5.19	12.28	**3.64**
CIFAR-10	50k+10k	3072	10	64.27	66.01	**61.15**

Auto-WEKA better than full grid search in 15/21 cases

Further details: KDD-13 paper (to appear)

PbO enables ...

- performance optimisation for different use contexts
  (some details later)
- adaptation to changing use contexts
  (see, *e.g.*, life-long learning – Thrun 1996)
- self-adaptation while solving given problem instance
  (*e.g.*, Battiti *et al.* 2008; Carchrae & Beck 2005; Da Costa *et al.* 2008)
- automated generation of instance-based solver selectors
  (*e.g.*, SATzilla – Leyton-Brown *et al.* 2003, Xu *et al.* 2008;
  Hydra – Xu *et al.* 2010; ISAC – Kadioglu *et al.* 2010)
- automated generation of parallel solver portfolios
  (*e.g.*, Huberman *et al.* 1997; Gomes & Selman 2001;
  Schneider *et al.* 2012)

Design space specification

Option 1: use language-specific mechanisms

- command-line parameters
- conditional execution
- conditional compilation (`ifdef`)

Option 2: generic programming language extension

Dedicated support for ...

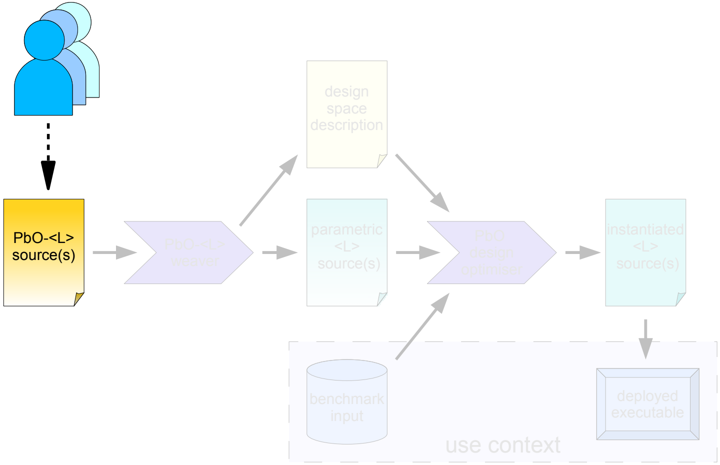
- exposing parameters
- specifying alternative blocks of code

Advantages of generic language extension:

- reduced overhead for programmer
- clean separation of design choices from other code
- dedicated PbO support in software development environments

Key idea:

- augmented sources: *PbO-Java* = Java + PbO constructs, ...
- tool to compile down into target language: *weaver*


PbO-<L>
source(s)
PbO-<L>
weaver
design
space
description
parametric
<L>
source(s)
PbO
design
optimiser
instantiated
<L>
source(s)
benchmark
input
deployed
executable
use context

Exposing parameters

```
...
numerator -= (int) (numerator / (adjfactor+1) * 1.4);
...      ...
##PARAM(float multiplier=1.4)
numerator -= (int) (numerator / (adjfactor+1) * ##multiplier);
...
```

- parameter declarations can appear at arbitrary places (before or after first use of parameter)
- access to parameters is read-only (values can only be set/changed via command-line or config file)

Specifying design alternatives

- **Choice:** set of interchangeable fragments of code that represent design alternatives (**instances of choice**)
- **Choice point:** location in a program at which a choice is available

```
##BEGIN CHOICE preProcessing
<block 1>
##END CHOICE preProcessing
```

Specifying design alternatives

- **Choice:** set of interchangeable fragments of code that represent design alternatives (**instances of choice**)

- **Choice point:**
  location in a program at which a choice is available

```
##BEGIN CHOICE preProcessing=standard
<block S>
##END CHOICE preProcessing

##BEGIN CHOICE preProcessing=enhanced
<block E>
##END CHOICE preProcessing
```

Specifying design alternatives

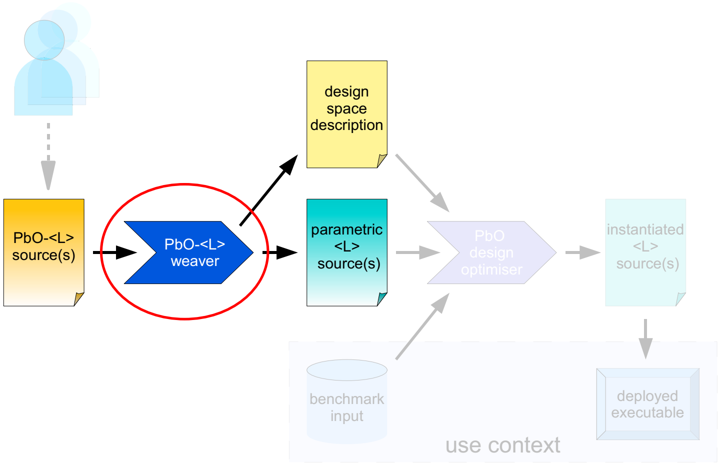
- **Choice:** set of interchangeable fragments of code that represent design alternatives (**instances of choice**)
- **Choice point:** location in a program at which a choice is available

```
##BEGIN CHOICE preProcessing
<block 1>
##END CHOICE preProcessing

...

##BEGIN CHOICE preProcessing
<block 2>
##END CHOICE preProcessing
```

Specifying design alternatives

- **Choice:** set of interchangeable fragments of code that represent design alternatives (**instances of choice**)
- **Choice point:** location in a program at which a choice is available

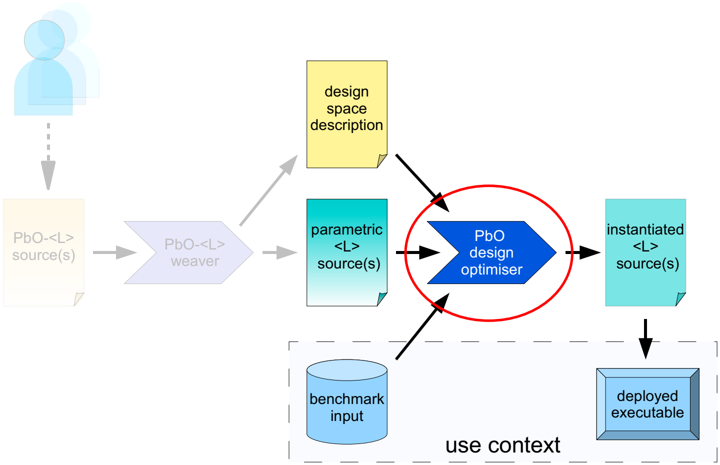
```
##BEGIN CHOICE preProcessing
<block 1a>
  ##BEGIN CHOICE extraPreProcessing
  <block 2>
  ##END CHOICE extraPreProcessing
<block 1b>
##END CHOICE preProcessing
```

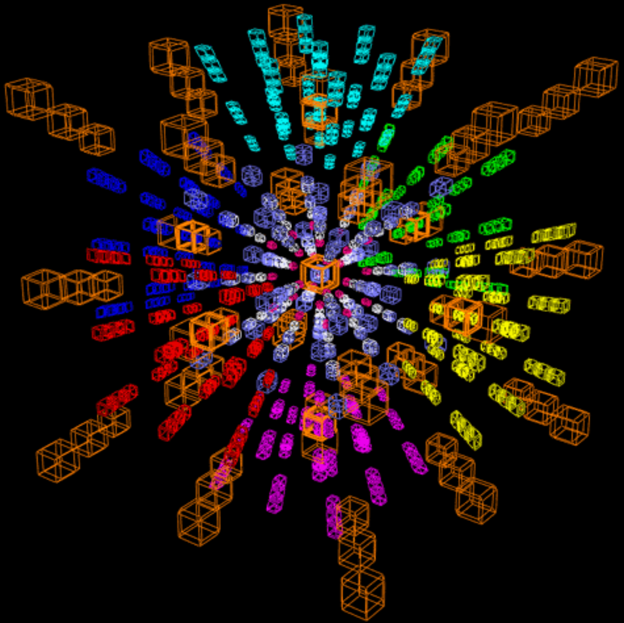

design
space
description
PbO-<L>
source(s)
PbO-<L>
weaver
parametric
<L>
source(s)
PbO
design
optimiser
instantiated
<L>
source(s)
benchmark
input
deployed
executable
use context

The Weaver

transforms PbO-<L> code into <L> code
(<L> = Java, C++, ...)

- parametric mode:
  - expose parameters
  - make choices accessible via (conditional, categorical) parameters
- (partial) instantiation mode:
  - hardwire (some) parameters into code (expose others)
  - hardwire (some) choices into code (make others accessible via parameters)


PbO-<L>
source(s)
PbO-<L>
weaver
design
space
description
parametric
<L>
source(s)
PbO
design
optimiser
instantiated
<L>
source(s)
benchmark
input
deployed
executable
use context



Design optimisation

Simplest case: Configuration / tuning

- Standard optimisation techniques

  (*e.g.*, CMA-ES – Hansen & Ostermeier 01; MADS – Audet & Orban 06)

- Advanced sampling methods

  (*e.g.*, REVAC, REVAC++ – Nannen & Eiben 06–09)

- Racing

  (*e.g.*, F-Race – Birattari, Stützle, Paquete, Varrentrapp 02;
  Iterative F-Race – Balaprakash, Birattari, Stützle 07)

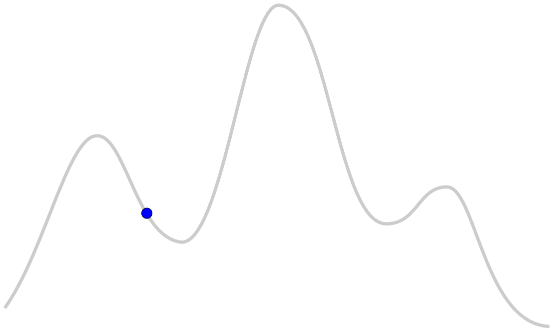
- Model-free search

  (*e.g.*, ParamILS – Hutter, HH, Stützle 07;
  Hutter, HH, Leyton-Brown, Stützle 09)

- Sequential model-based optimisation

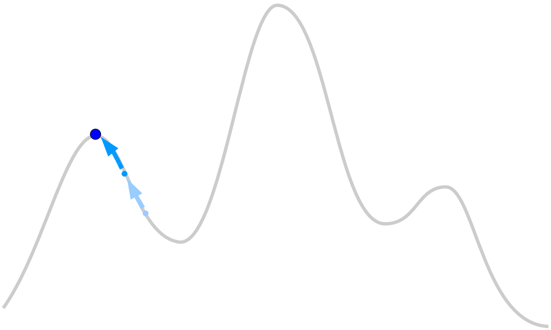
  (*e.g.*, SPO – Bartz-Beielstein 06; SMAC – Hutter, HH, Leyton-Brown 11–12)

Iterated Local Search



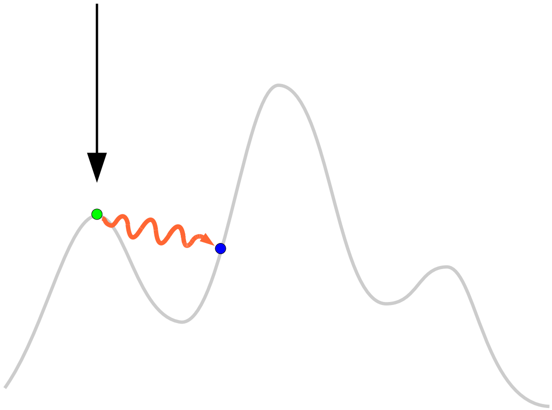
Initialisation

Iterated Local Search



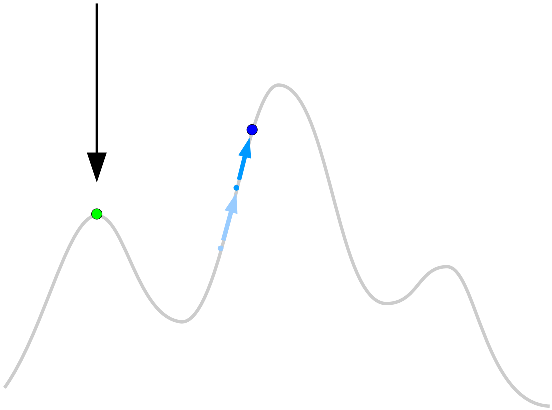
Local Search

Iterated Local Search



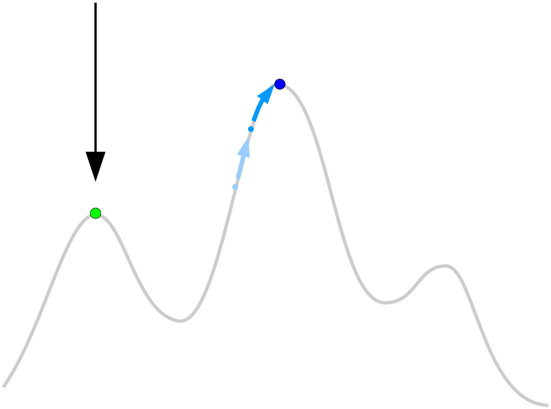
Perturbation

Iterated Local Search



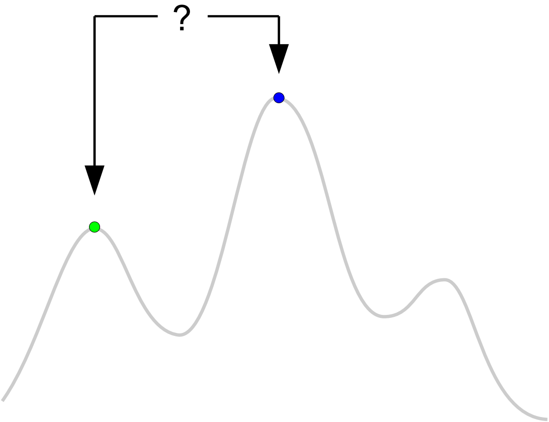
Local Search

Iterated Local Search



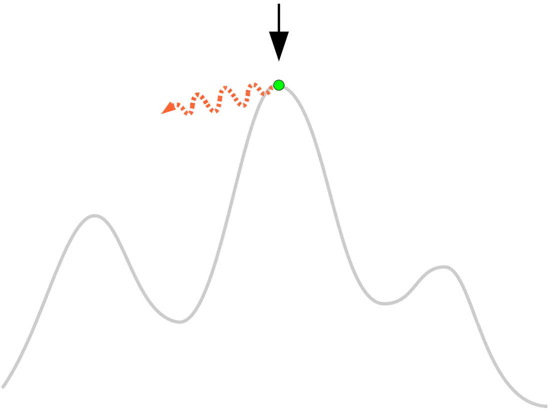
Local Search

Iterated Local Search



Selection (using Acceptance Criterion)

Iterated Local Search



Perturbation

ParamILS

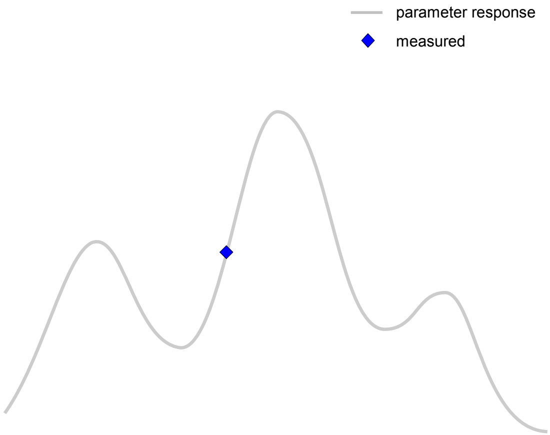
- iterated local search in configuration space
- initialisation: pick *best* of default + R random configurations
- subsidiary local search: iterative first improvement, change one parameter in each step
- perturbation: change s randomly chosen parameters
- acceptance criterion: always select *better* configuration
- number of runs per configuration increases over time; ensure that incumbent always has same number of runs as challengers

Sequential Model-based Optimisation

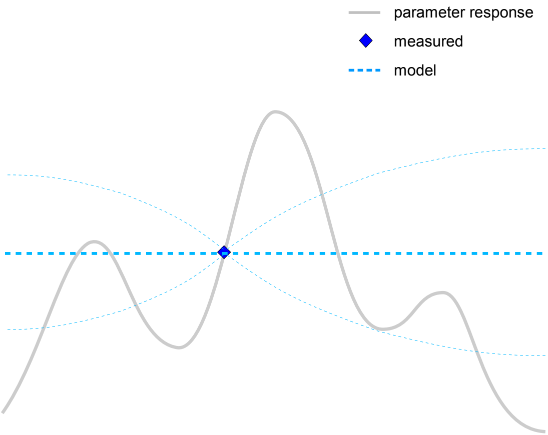
e.g., Jones (1998), Bartz-Beielstein (2006)

- **Key idea:**
  use predictive performance model (response surface model) to find good configurations
- perform runs for selected configurations (initial design) and fit model (*e.g.*, noise-free Gaussian process model)
- iteratively select promising configuration, perform run and update model

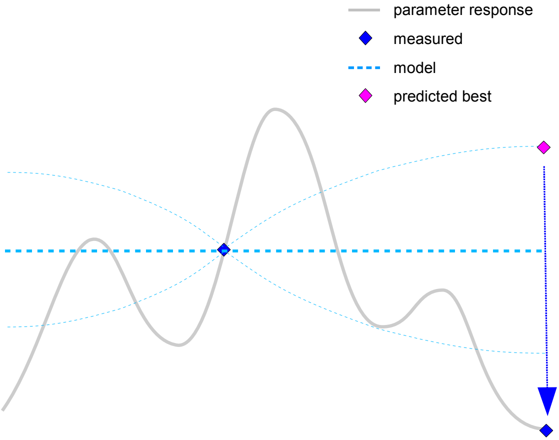
Sequential Model-based Optimisation



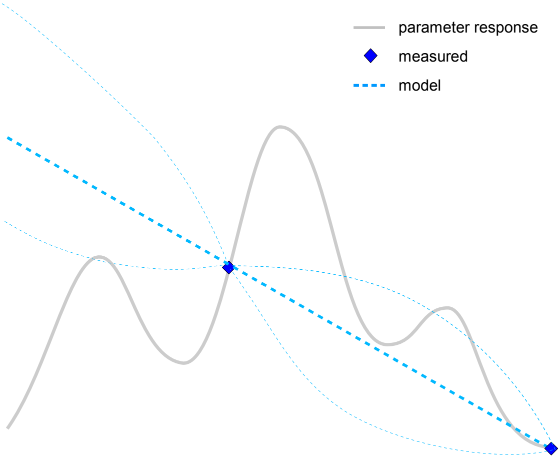
Sequential Model-based Optimisation



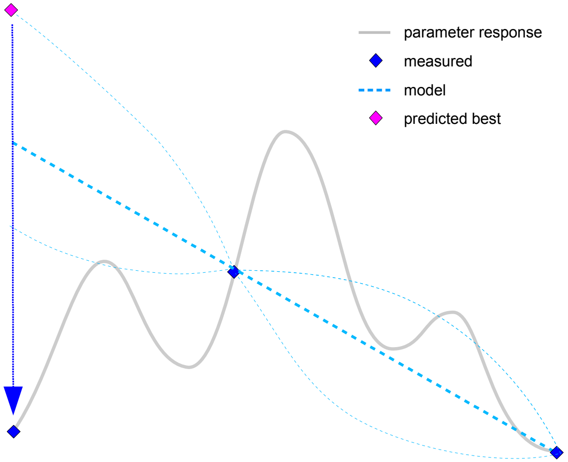
Sequential Model-based Optimisation



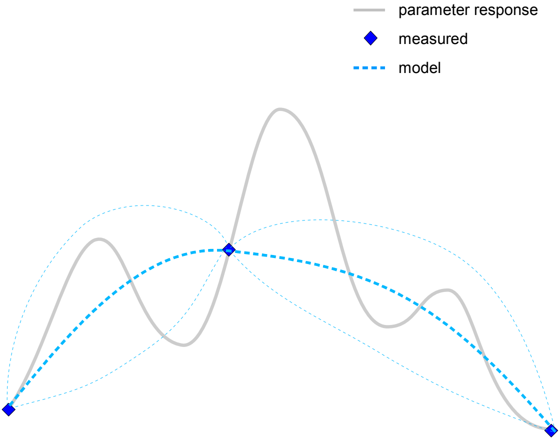
Sequential Model-based Optimisation



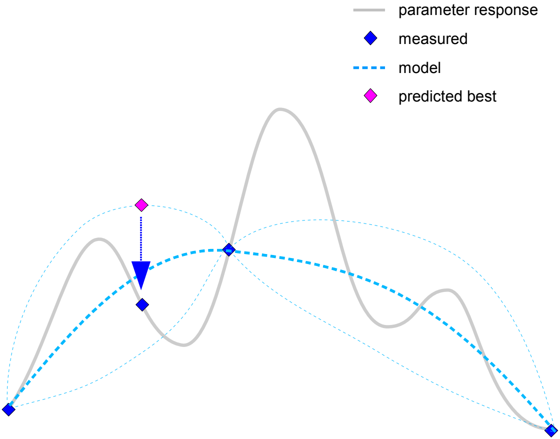
Sequential Model-based Optimisation



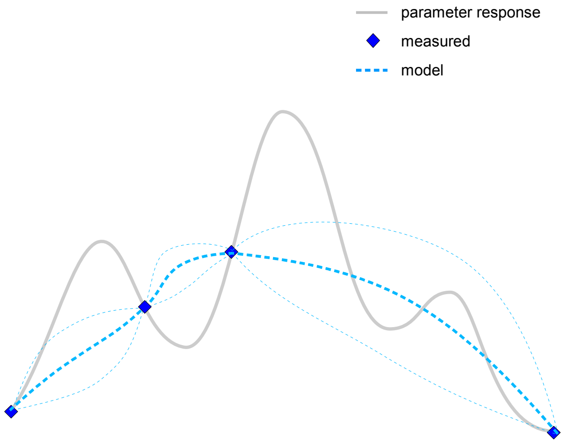
Sequential Model-based Optimisation



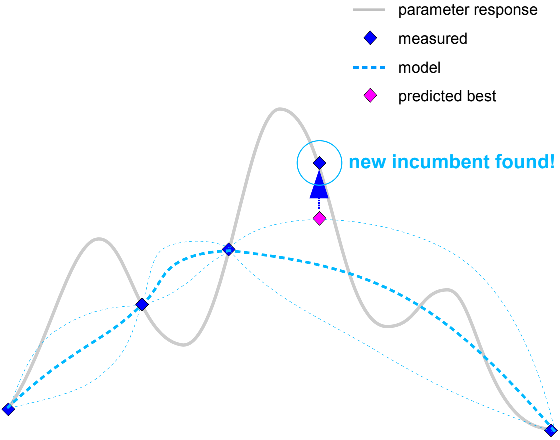
Sequential Model-based Optimisation



Sequential Model-based Optimisation



Sequential Model-based Optimisation



Sequential Model-based Algorithm Configuration (SMAC)

Hutter, HH, Leyton-Brown (2011)

- uses *random forest model* to predict performance of parameter configurations
- predictions based on algorithm parameters and instance features, aggregated across instances
- finds promising configurations based on *expected improvement criterion*, using multi-start local search and random sampling
- initialisation with single configuration (algorithm default or randomly chosen)

Effective use of automated configurators (for running time minimisation)

- $\geq 75\%$ of training set solvable by default configuration within cutoff time t
- avoid training instances that are too easy (< 0.1 CPU sec)
- the overall time budget per configurator run $\geq 200 \cdot t$ (better $\geq 1000 \cdot t$)
- conduct 10–25 independent configurator runs;
  evaluate resulting configurations on entire training set;
  select best

More on benchmark sets for automated configuration & evaluation of solvers:
HH, Kaufmann, Schaub, Schneider (2013)

Configuration for scaling performance

Styles & HH (2012–13)

Challenge:

Configure a given algorithm for good performance on instances too difficult to permit many evaluations

$\rightsquigarrow$ cannot directly use standard protocol

$\rightsquigarrow$ configure on easier inputs in a way that generalises to harder ones (= transfer learning)

Key idea:

- configure on easy instances (multiple configurator runs)
- select by validation on successively harder instances
- use racing (ordered permutation races) for efficient validation

Cost & concerns

But what about ...

- Computational complexity?
- Cost of development?
- Limitations of scope?

Computationally too expensive?

Spear revisited:

- total configuration time on software verification benchmarks: $\approx$ 30 CPU days
- wall-clock time on 10 CPU cluster: $\approx$ 3 days
- cost on Amazon Elastic Compute Cloud (EC2): 61.20 USD (= 42.58 EUR)
- 61.20 USD pays for ...
  - 1:45 hours of average software engineer
  - 8:26 hours at minimum wage

Too expensive in terms of development?

Design and coding:

- tradeoff between performance/flexibility and overhead
- overhead depends on level of PbO
- traditional approach: cost from manual exploration of design choices!

Testing and debugging:

- design alternatives for individual mechanisms and components can be tested separately

$\rightsquigarrow$ effort linear (rather than exponential) in the number of design choices

Limited to the "niche" of NP-hard problem solving?

Some PbO-flavoured work in the literature:

- computing-platform-specific performance optimisation of linear algebra routines

  (Whaley *et al.* 2001)

- optimisation of sorting algorithms using genetic programming

  (Li *et al.* 2005)

- compiler optimisation

  (Pan & Eigenmann 2006, Cavazos *et al.* 2007)

- database server configuration

  (Diao *et al.* 2003)

The road ahead

- Support for PbO-based software development
  - Weavers for PbO-C, PbO-C++, PbO-Java
  - PbO-aware development platforms
  - Improved / integrated PbO design optimiser
- Best practices
- Many further applications
- Scientific insights

Leveraging parallelism

- design choices in parallel programs
  (Hamadi, Jabhour, Sais 2009)
- deriving parallel programs from sequential sources
  $\leadsto$ concurrent execution of optimised designs
  (parallel portfolios)
  (HH, Leyton-Brown, Schaub, Schneider 2012)
- parallel design optimisers
  (*e.g.*, Hutter, Hoos, Leyton-Brown 2012)

Which choices matter?

Observation: Some design choices matter more than others

depending on ...

- algorithm under consideration
- given use context

Knowledge which choices / parameters matter may ...

- guide algorithm development
- facilitate configuration

3 recent approaches:

- Forward selection based on empirical performance models
  Hutter, HH, Leyton-Brown (2013)
- Functional ANOVA based on empirical performance models
  Hutter, HH, Leyton-Brown (under review)
- Ablation analysis
  Fawcett, HH (to appear)

Ablation analysis

Fawcett, HH (to appear)

Key idea:

- given two configurations, A and B, change one parameter at a time to get from A to B

  $\rightsquigarrow$ ablation path

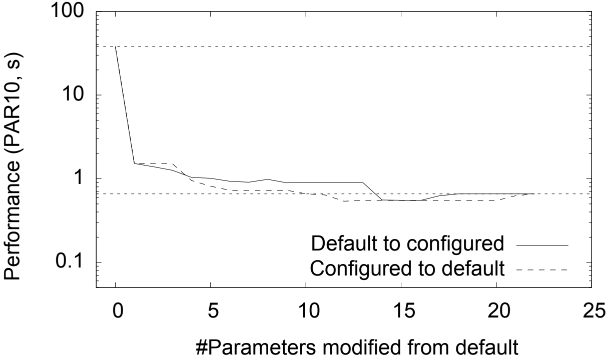
- in each step, change parameter to achieve maximal gain (or minimal loss) in performance

- for computational efficiency, use racing (F-race) for evaluating parameters considered in each step

Empirical study:

- high-performance solvers for SAT, MIP, AI Planning
  (26–76 parameters),
  well-known sets of benchmark data (real-world structure)
- optimised configurations obtained from ParamILS
  (minimisation of penalised average running time;
  10 runs per scenario, 48 CPU hours each)

Ablation between default and optimised configurations:



LPG on Depots planning domain

Which parameters are important?

LPG on `depots`:

- `cri_intermediate_levels` (43% of overall gain!)
- `triomemory`
- `donot_try_suspected_actions`
- `walkplan`
- `weight_mutex_in_relaxed_plan`

Note: Importance of parameters varies between planning domains

Programming by Optimisation ...

- leverages computational power to construct better software
- enables creative thinking about design alternatives
- produces better performing, more flexible software
- facilitates scientific insights into
  - efficacy of algorithms and their components
  - empirical complexity of computational problems

... changes how we build and use high-performance software

Acknowledgements

Collaborators:

- Domagoj Babić
- Chris Fawcett
- Quinn Hsu
- Frank Hutter
- Erez Karpas
- Chris Nell
- Steve Ramage
- Gabriele Röger
- Marius Schneider
- James Styles
- Chris Thornton
- Mauro Vallati
- Lin Xu

- Thomas Bartz-Beielstein
  (FH Köln, Germany)
- Marco Chiarandini
  (Syddansk Universitet, Denmark)
- Alfonso Gerevini
  (Università degli Studi di Brescia, Italy)
- Malte Helmert
  (Universität Basel, Switzerland)
- Alan Hu
- Kevin Leyton-Brown
- Kevin Murphy
- Alessandro Saetti
  (Università degli Studi di Brescia, Italy)
- Torsten Schaub
  (Universität Potsdam, Germany)
- Thomas Stützle
  (Université Libre de Bruxelles, Belgium)

Research funding:

- NSERC, Mprime, GRAND, CFI
- IBM, Actenum Corp.

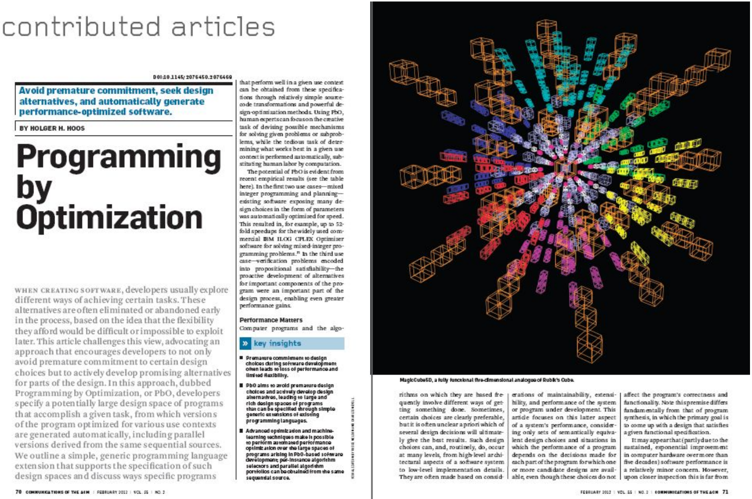
Computing resources:

- Arrow, BETA, ICICS clusters
- Compute Canada / WestGrid

Gli uomini hanno idee [...]
– Le idee, se sono allo stato puro, sono belle.
Ma sono un meraviglioso casino.
Sono apparizioni provvisorie di infinito.

People have ideas [...]
– Ideas, in their pure state, are beautiful.
But they are an amazing mess.
They are fleeting apparitions of the infinite.

(Prof. Mondrian Kilroy *in Alessandro Baricco: City*)



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 algorithms 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 considerations 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

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.

Communications of the ACM, 55(2), pp. 70–80, February 2012

www.prog-by-opt.net