# Computer-Aided Design of High-Performance Algorithms:

# Principled Procedures for Building Better Solvers

Holger H. Hoos

BETA Lab
Department of Computer Science
University of British Columbia
Canada

LION 4 Invited Tutorial

# High-performance heuristic algorithms are difficult to design

- many design choices (representation / search space; neighbourhoods; search strategy; variable/value selection heuristic; restart rules; pre-processing; data structures; ...)

- best performance often achieved by combination of various heuristics
  (e.g., best improvement + random restart, multi-phase search, systematic search + preprocessing, iterated local search, local + systematic search)

- various heuristic components interact in complex ways
  $\rightsquigarrow$ unexpected, emergent behaviour

- performance can be tricky to assess due to
  - differences in behaviour across problem instances
  - stochasticity

## Therefore ...

- time-consuming design process, success often critically dependent on experience, intuition, luck
- resulting algorithms often complex, somewhat ad-hoc, not fully optimised

## Real-world example:

- Application: Solving SAT-encoded software verification problems
- Given: High-performance DPLL-type SAT solver (SPEAR)
  - 26 parameters (7 categorical, 3 Boolean, 12 continuous, 4 integer-valued)
  - control variable/value ordering heuristics, clause learning, restarts, ...
- Goal: Minimize expected run-time on 'typical' SAT instances from software verification tool
- Problems:
  - default settings $\rightsquigarrow$ $\approx$ 300 seconds / run
  - good performance on a few instances may not generalise

# Outline

1. Introduction
2. From traditional to computer-aided algorithm design
3. Design spaces and design patterns
4. Meta-algorithmic search and optimisation procedures
5. Selected computer-aided algorithm design procedures
6. Three success stories
7. Towards a software environment for computer-aided design of high-performance algorithms

# From traditional to computer-aided algorithm design

Traditional algorithm design approach:

- iterative, manual process
- designer gradually introduces/modifies components or mechanisms
- test performance on benchmark instances
- design often starts from generic or broadly applicable problem solving method (e.g., evolutionary algorithm)

Note:

- During the design process, many decisions are made.
- Some choices take the form of parameters, others are hard-coded.
- Design decisions interact in complex ways.

## Problems:

- Design process is labour-intensive.
- Design decisions often made in *ad-hoc* fasion, based on limited experimentation and intuition.
- Human designers typically over-generalise observations, explore few designs.
- Implicit assumptions of independence, monotonicity are often incorrect.
- Number of components and mechanisms tends to grow in each stage of design process.

$\leadsto$ complicated designs, unfulfilled performance potential

## Solution: Computer-aided Algorithm Design

- Goal: construct high-performance algorithms automatically
- Key idea: use fully formalised procedures to effectively explore large space of candidate designs

$\rightsquigarrow$ genetic programming, hyper-heuristics, learning and intelligent optimisation, SLS engineering

## Human designer:

- specifies (possibly large) space of candidate algorithm design
- supplies set of problem instances for performance evaluation
- specifies performance metric

## Meta-algorithmic system:

- explores design space in principled manner
- evaluates candidate design
- finds high-performance designs

## Advantages:

- lets human designer focus on higher-level issues
- enables better exploration of larger design spaces
- exploits complementary strengths of different approaches for solving a given problem
- uses principled, fully formalised methods for algorithm design
- can be used to customise algorithms for use in specific applications with minimal human effort

## Example: SAT-based software verification

Hutter, Babic, Hoos, Hu – FMCAD'07

- **Goal:** Solve suite of SAT-encoded software verification instances as fast as possible
- new DPLL-style SAT solver SPEAR (by Domagoj Babic)
  = 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, Hoos, Stützle – AAAI'07]

## SPEAR: Empirical results 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)

# Design spaces and design patterns

Special cases of computer-aided algorithm design:

- parameter optimisation (for given set of instances)
  Birattari et al. 2002; Adenso-Diaz & Laguna 2006, Hutter et al. 2007; Bartz-Beielstein 2006
- algorithm configuration from components
  (for given set of instances)
  Fukunaga 2002, Chiarandini et al. 2008, KhudaBukhsh et al. 2009
- restart strategies
  Luby et al. 1993; Gagliolo & Schmidhuber 2007

## Special cases of computer-aided algorithm design (2):

- instance-based algorithm configurators
  Hutter et al. 2006

- on-line algorithm control / reactive search
  Carchrae & Beck 2005; Battiti et al. 2008

- instance-based algorithm selection
  Rice 1976; Leyton-Brown et al. 2003; Guerri & Milano 2004;
  Xu et al. 2008

- algorithm portfolios (static and dynamic)
  Huberman et al. 1997, Gomes & Selman 2001;
  Gagliolo & Schmidhuber 2007

$\rightsquigarrow$ meta-algorithmic design patterns, induce design spaces

## Generalised local search machines (GLSMs)

Hoos (1998); Hoos & Stützle (2004)

- formal model for complex / hybrid stochastic local search (SLS) algorithms
- facilitate the design of complex SLS algorithms by *structuring* and *restricting* the design space
- abstract GLSMs as instantiable design patterns
  - capture structure of search control mechanism
  - instantiation of state and transition types results in SLS algorithm

# Meta-algorithmic search and optimisation procedures

How to search design spaces?

- use powerful heuristic search and optimisation procedures, combined with significant amounts of computing power
- use machine learning methods (classification, regression), combined with significant amount of training data

## Some examples:

- parameter tuning:
  - numerical optimisation techniques
    e.g., CMA-ES (Hansen & Ostermeier 2001)
  - model-based optimisation methods
    e.g., SPO (Bartz-Beielstein 2006),
    SPO$^+$ (Hutter et al. 2009)
- algorithm configuration:
  - genetic programming
    e.g., CLASS (Fukunaga 2002)
  - racing procedures
    e.g., F-Race (Birattari et al. 2002)
  - advanced stochastic local search procedures
    e.g., ParamILS (Hutter et al. 2007)

## More examples:

- instance-based algorithm selection
  - classification approaches (e.g., Guerri & Milano 2004)
  - regression approaches (e.g., Leyton-Brown et al. 2003, Xu et al. 2008)
- dynamic algorithm portfolios (time allocators)
  - bandit solvers (e.g., Gagliolo & Schmidhuber 2007)
  - evolutionary algorithms (e.g., Harick & Lobo 1999)

## Many open questions:

- Which procedure for which type of design space?
- How to deal with hybrid design patterns?
- How to best deal with censored, sparse data?

# Selected computer-aided algorithm design procedures:

- F-Race / Iterative F-Race
  Birattari, Stützle, Paquete, Varrentrapp (2002);
  Balaprakash, Birattari, Stützle (2007)

- ParamILS
  Hutter, Hoos, Stützle (2007); Hutter, Hoos, Leyton-Brown, Stützle (2009)

- SPO / SPO$^+$
  Bartz-Beielstein (2006); Hutter, Hoos, Leyton-Brown, Murphy (2009)

- SATzilla
  Xu, Hutter, Hoos, Leyton-Brown (2008)

# Automated algorithm selection/configuration using F-Race

Birattari, Stützle, Paquete, Varrentrapp (2002); Balaprakash, Birattari, Stützle (2007)

Key idea:

- *Given:* set $S$ of algorithms for a problem, set of problem instances $\Pi$
- *Select* from $S$ the algorithm expected to solve instances from $\Pi$ *most efficiently* on average

## F-Race (Birattari, Stützle, Paquete, Varrentrapp 2002)

- inspired by methods for model selection methods in machine learning
  (Maron & Moore 1994; Moore & Lee 1994)
- sequentially evaluate algorithms/configuration, in each iteration, perform one new run per algorithm/configuration
- eliminate poorly performing algorithms/configurations as soon as sufficient evidence is gathered against them
- use Friedman test to detect poorly performing algorithms/configurations

# Iterative F-Race (Balaprakash, Birattari, Stützle 2007)

**Problem:** When using F-Race for algorithm configuration, number of initial configurations considered is severely limited.

**Solution:**

- perform *multiple iterations of F-Race* on *limited set of configurations*
- sample candidate configurations based on *probabilistic model* (independent normal distributions centred on surviving configurations)
- gradually reduce variance over iterations (*volume reduction*)

$\leadsto$ good results for

- MAX-MIN Ant System for the TSP (6 parameters)
- simulated annealing for stochastic vehicle routing (4 parameters)
- estimation-based local search for PTSP (3 parameters)

# Automated algorithm configuration using ParamILS

Hutter, Hoos, Stützle (2007); Hutter, Hoos, Leyton-Brown, Stützle (2009)

## Key idea:

- *Given:* parameterised algorithm $A$ for a problem, set of problem instances $\Pi$
- *Select* parameter values of $A$ to solve instances from $\Pi$ *most efficiently* based on *search in configuration space*

**Goal:** Apply to algorithms with

- many parameters, relatively few instances.
- categorical parameters

## ParamILS (Hutter, Hoos, Stützle 2007)

- initialisation: pick *best* of default + $R$ random configurations
- iterated local search in configuration space
- 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 'new' configurations

## Some example applications:

- SLS for 2D/3D HP protein structure prediction (5 parameters)
  Thachuk, Shmygelska, Hoos (2007)
- DPLL for SAT-encoded software verification (26 parameters)
  Hutter, Babic, Hoos, Hu (2007)
- CPLEX for mixed integer programming (63 parameters)
  Hutter, Hoos, Leyton-Brown, Stützle (2009)
- University timetabling (7+11 parameters)
  Chiarandini, Fawcett, Hoos (2008); Fawcett, Chiarandini, Hoos (2009)

$\leadsto$ substantial improvements in state of the art for solving these (and other) problems

# Model-based parameter tuning using Sequential Parameter Optimisation

Bartz-Beielstein (2006); Hutter, Hoos, Leyton-Brown (2009)

Key idea:

- *Given:* parameterised algorithm $A$ for a problem, set of problem instances $\Pi$
- *Select* parameter values of $A$ to solve instances from $\Pi$ *most efficiently* based on *predictive performance model*

## Sequential Parameter Optimisation (SPO):

Bartz-Beielstein (2006)

- perform runs for selected configurations (initial design) and fit (noise-free Gaussian process) model
- iteratively select promising configuration $C$, run $A$ using $C$ and update model

- initial design: *Latin Hypercube Design (LHD)*
- use *expected improvement criterion* to select promising configurations
- *intensification mechanism:*
  - gradually increase number of runs for each configuration;
  - ensure that incumbent always has same number of runs as 'new' configurations

## Latest variant: SPO$^+$ (Hutter, Hoos, Leyton-Brown, Murphy 2009)

- model/predict *log-transformed* performance data
- modified intensification mechanism ensures that sufficiently many runs are performed before changing incumbent

## Example applications:

- CMA-ES (state-of-the-art continuous optimisation procedure)
- SAPS (high-performance SAT algorithm)

$\leadsto$ substantial performance improvements over default configurations

(Ongoing work on better handling of tuning over multiple instances.)

# Instance-specific algorithm selection based on run-time predictions

Leyton-Brown, Nudelman, Shoham (2002); Xu, Hutter, Hoos, Leyton-Brown (2008)

Key idea: (Rice 1976)

- *Given:* set $S$ of algorithms for a problem, problem instance $\pi$
- *Select* from $S$ the algorithm expected to solve $\pi$ *most efficiently*, based on (cheaply computable) *features* of $\pi$

Here:

- problem instance $\rightsquigarrow$ vector of cheaply computable features
- features $\rightsquigarrow$ performance prediction for given set of solvers
- run solver with best predicted performance

## Instance features:

- Use generic and problem-specific features that correlate with performance and can be computed cheaply.
- Examples (for SAT):
  - number of clauses, variables, ...
  - constraint graph features
  - local & complete search probes
- Use as features statistics of distributions, e.g., variation coefficient of node degree in constraint graph
- Consider pairwise products of features (quadratic basis function expansion).

## Run-time prediction:

- Collect feature and performance data on (large & diverse ) set of training instances.
- Use feature selection to avoid problems due to correlated / uninformative features.
- Use ridge regression on training data to build predictive model.

## Example applications:

- SATzilla (Leyton-Brown et al. 2002; Xu et al. 2008)
- Winner determination for combinatorial auctions (Leyton-Brown et al. 2003, 2009)

# Three success stories

How good are current methods for computer-aided algorithm design?

**"The proof is in the pudding":**

- Propositional Satisfiability
- Course Timetabling
- Mixed Integer Programming (CPLEX)

Further successes:

- protein structure prediction (Thachuk et al. 2007)
- TSP (Styles & Hoos *in preparation*)
- real-world scheduling for the oil & gas industry (Actenum Corp.)

# SATenstein: Automatically Building Local Search Solvers for SAT

KhudaBukhsh, Xu, Hoos, Leyton-Brown – IJCAI-09

**Frankenstein:**

create perfect human being from scavenged body parts

**SATenstein:**

create perfect SAT solvers using components scavenged from existing solvers

**Geneneral approach:**

- components from GSAT, WalkSAT, dynamic local search and G2WSAT algorithms
- flexible SLS framework (derived from UBCSAT)
- find performance-optimising instantiations using ParamILS

**Challenge:**

- 41 parameters (mostly categorical)
- over $2 \cdot 10^{11}$ configurations

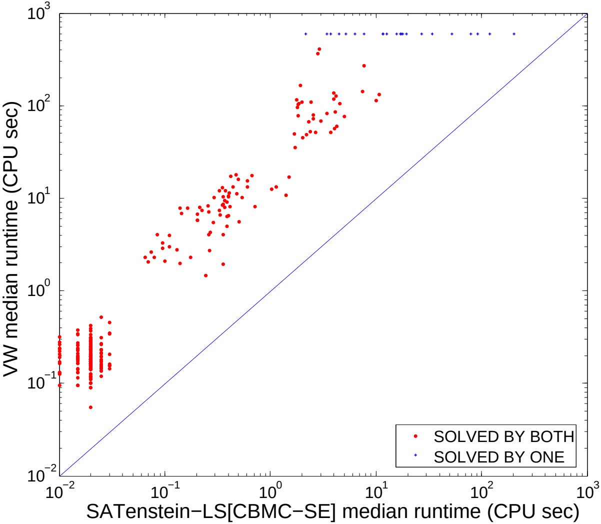
- 6 well-known distributions of SAT instances
  (QCP, SW-GCP, R3SAT, HGEN, FAC, CBMC-SE)

- 11 challenger algorithms
  (includes all winning SLS solvers from SAT competitions 2003–2008)

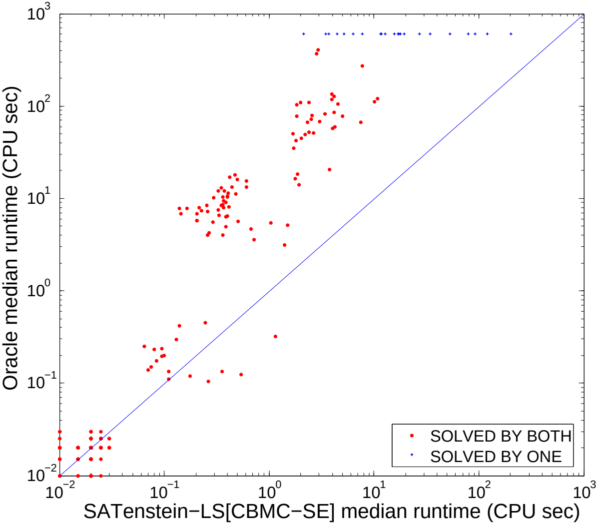
**Result:**

- factor 70–1000 performance improvements over best challengers on QCP, HGEN, CBMC-SE
- factor 1.4–2 performance improvement over best challengers on SW-GCP, R3SAT, FAC

# SATenstein-LS vs VW on CBMC-SE



# SATenstein-LS vs Oracle on CBMC-SE



# SATzilla: Portfolio-based algorithm selection for SAT

Xu, Hutter, Hoos, Leyton-Brown (JAIR 2008)

## Key idea: Instance-based Algorithm Selection

- *Given:* set $S$ of high-performance SAT solvers (DPLL and SLS), CNF formula $F$
- *Select* the algorithm from $S$ expected to solve $F$ *most efficiently*, based on (cheaply computable) *features* of $F$.

## SATzilla in a nutshell:

- CNF formula $\rightsquigarrow$ 84 polytime-computable instance features
- features $\rightsquigarrow$ performance prediction for set of SAT solvers
- run solver with best predicted performance

## Under the hood:

- Use state-of-the-art complete (DPLL) and incomplete (local search) SAT solvers.
- Use ridge regression on selected features to predict solver run-times from instance features.
- Use method by Schmee & Hahn (1979) to deal with censored run-time data.

## Some bells and whistles:

- Use pre-solvers to solve 'easy' instances quickly.
- Build run-time predictors for various types of instances, use classifier to select best predictor based on instance features.
- Predict time required for feature computation; if that time is too long, use back-up solver.

$\rightsquigarrow$ prizes in 5 of the 9 main categories of the 2009 SAT Solver Competition (3 gold, 2 silver medals)

# Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection

Xu, Hoos, Leyton-Brown – work in progress

**Note:**

- SATenstein builds solvers that work well on average on a given set of SAT instances
  *but:* may have to settle for compromises for broad, heterogenous sets

- SATzilla builds algorithm selector based on given set of SAT solvers
  *but:* success entirely depends on quality of given solvers

**Idea:** Combine the two approaches $\rightsquigarrow$ portfolio-based selection from set of automatically constructed solvers

## Simple combination:

1. build solvers for various types of instances using automated algorithm configuration
2. construct portfolio-based selector from these

**Drawback:** Requires suitably defined sets of instances

## Better solution:

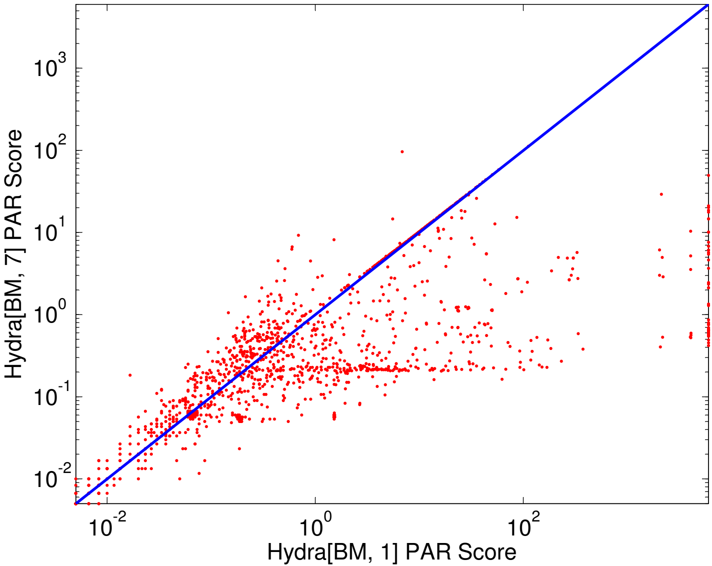
iteratively build & add solvers that improve performance of given portfolio

$\rightsquigarrow$ **Hydra**

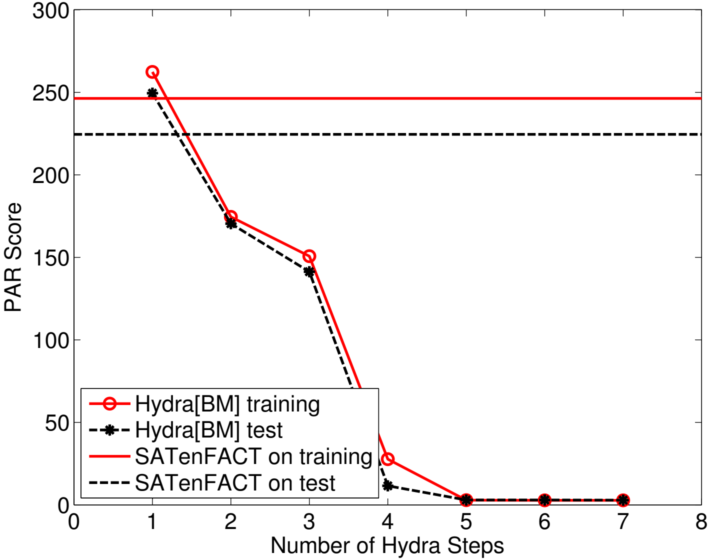
**Note:** Builds portfolios solely using

- generic, highly configurable solver (e.g., SATenstein)
- features (as used in SATzilla)

# First results of Hydra for SAT



# First results of Hydra for SAT



# Post-Enrolment Course Timetabling

Chiarandini, Fawcett, Hoos (2008); Fawcett, Hoos, Chiarandini (in preparation)

## Post-Enrolment Course Timetabling:

- students enroll in courses
- courses are assigned to rooms and time slots, subject to *hard constraints*
- preferences are represented by *soft constraints*

## Our solver:

- modular multiphase stochastic local search algorithm
- hard constraint solver: finds feasible course schedules
- soft constraint solver: optimise schedule (maintaining feasibility)

## Our first solver:

- developed over ca. 1 month
- starting point: Chiarandini et al. (2003)
- *soft constraint solver* unchanged
- automatically configured *hard constraint solver*

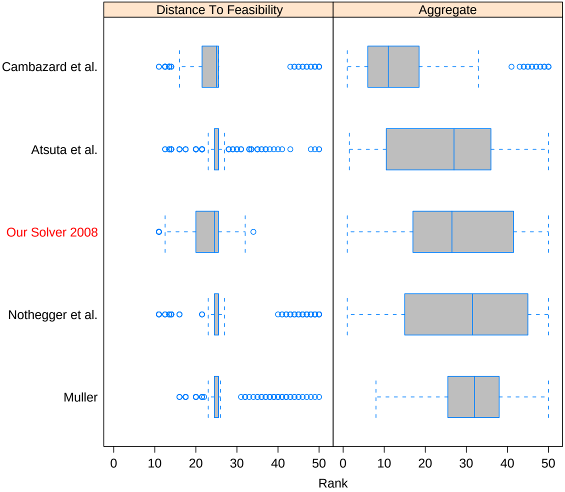
## Design space for hard constraint solver:

- parameterised combination of constructive search, tabu search, diversification strategy
- 7 parameters, 50 400 configurations

## Automated configuration process:

- configurator: FocusedILS 2.3 (Hutter et al. 2009)
- performance objective: solution quality after 300 CPU sec

# 2nd International Timetabling Competition (ITC), Track 2



## Our latest solver:

- developed over ca. 6 months
- starting point: our previous solver
- automatically configured *hard & soft constraint* solvers

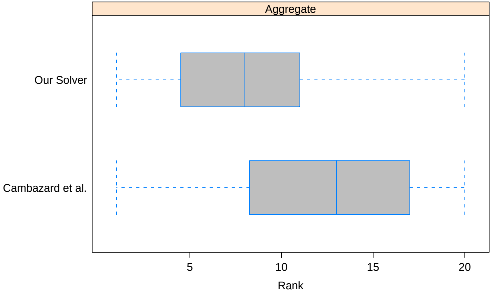
## Design space for soft constraint solver:

- highly parameterised simulated annealing algorithm
- 11 parameters, $2.7 \times 10^9$ configurations

## Automated configuration process:

- configurator: FocusedILS 2.4 (new version, multiple stages)
- multiple performance objectives
  (final stage: solution quality after 600 CPU sec)

# 2-way race against ITC Track 2 winner




- our solver wins beats ITC winner on 20 out of 24 competition instances
- application to university-wide exam scheduling at UBC in 2010

# Mixed Integer Programming (MIP)

Hutter, Hoos, Leyton-Brown, Stützle (2009)

- 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 10.0 user manual, p.247]

## Automatically Configuring CPLEX:

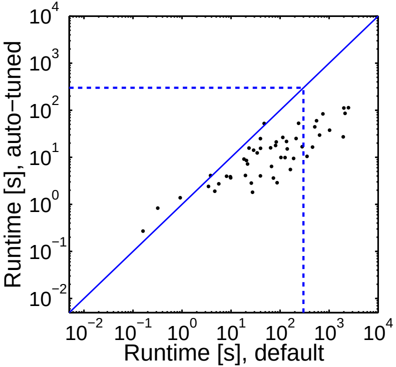
- starting point: factory default settings
- 63 parameters (some with 'AUTO' settings)
- $1.38 \times 10^{37}$ configurations
- configurator: FocusedILS 2.3 (Hutter et al. 2009)
- performance objective: minimal mean run-time
- configuration time: $10 \times 2$ CPU days

## CPLEX on various MIPS benchmarks

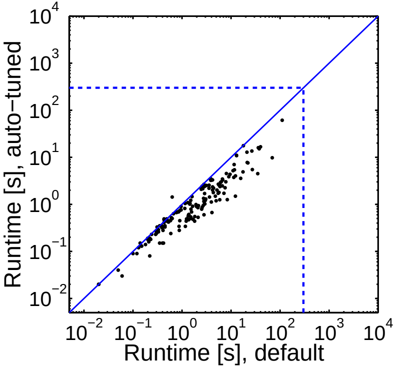
| Benchmark | Default performance [CPU sec] | Optimised performance [CPU sec] | Improvement factor |
|---|---|---|---|
| BCOL/CONIC.SCH | 5.37 | 2.35 ($2.4 \pm 0.29$) | 2.2 |
| BCOL/CLS | 712 | 23.4 ($327 \pm 860$) | 30.4 |
| BCOL/MIK | 64.8 | 1.19 ($301 \pm 948$) | 54.4 |
| CATS/REGIONS200 | 72 | 10.5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

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

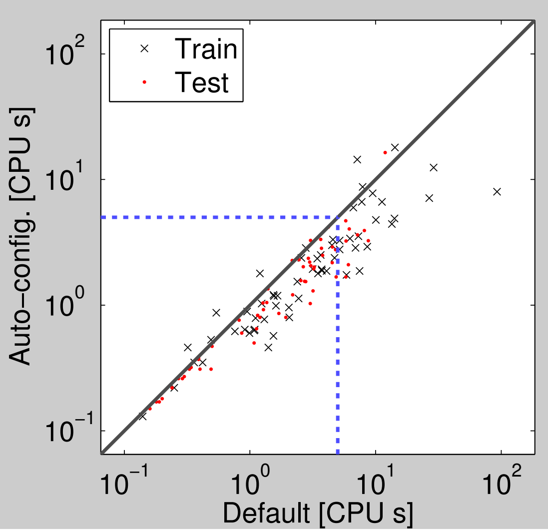
# CPLEX on BCOL/CLS



# CPLEX on BCOL/Conic.sch

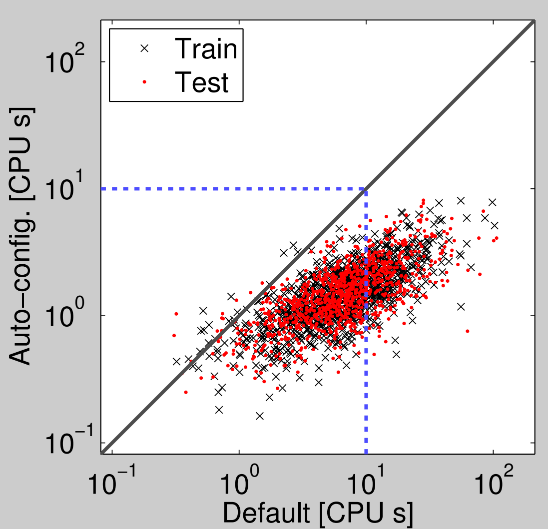


# Latest results: Gurobi on BCOL-MIK




Configuration time: $10 \times 1$h of ParamILS

# Latest results: lpsolve on CA-WDP




Configuration time: $10 \times 2$h of ParamILS

# Towards a software environment for computer-aided design of high-performance algorithms

How to best support use of computer-aided algorithm design methods?

- develop powerful procedures for *searching* large design spaces
- develop useful abstractions for *specifying* design spaces
- develop best practices for computer-aided algorithm design
- provide comfortable software environments for computer-aided algorithm design

## HAL: High-performance Algorithm Lab

Nell, Fawcett, Hoos, Leyton-Brown (work in progress)

- support *algorithm design* and *empirical analysis*
- support wide range of design patterns, procedures
- support effective utilisation of parallel computation
- support multiple platforms
  (Linux, Windows, MacOS, Chrome)
- web-based UI, component-based architecture
- open source, easy to use & expand

## A first implementation: HAL 1.0 (sneak preview)

- focus on automated algorithm configuration: ParamILS [Hutter et al.], GGA [Ansótegui et al.]
- empirical analysis of single solvers, pairs of solvers (comparative analysis)
- statistical tests (via R), plotting (via Gnuplot)
- system runs on Linux, Mac OS; web-based UI runs on any browser
- support for compute clusters, batch systems (pre-configured for Sun Grid Engine)

# Computer-aided Algorithm Design ...

- leverages computational power to construct better algorithms
- liberates human designers from boring, menial tasks and let them focus on higher-level design issues
- enables effective exploration of larger design spaces
- facilitates principled design of heuristic algorithms
- revolutionises the way we build and use algorithms

# Acknowledgements


**Collaborators:**

- Domagoj Babic
- Alex Devkar
- Chris Fawcett
- Ashiqur KhudaBukhsh
- Frank Hutter
- Chris Nell
- Eugene Nudelman
- Alena Shmygelska
- Chris Thachuk
- James Styles
- Lin Xu
- Thomas Bartz-Beielstein
  (FH Köln)
- Marco Chiarandini
  (University of Southern Denmark)
- Alan Hu
- Kevin Leyton-Brown
- Kevin Murphy
- Yoav Shoham
  (Stanford University)
- Thomas Stützle
  (Université Libre de Bruxelles)

**Research funding:**

- NSERC, MITACS, CFI
- IBM, Actenum Corp.

**Computing resources:**

- Arrow, BETA, ICICS clusters
- WestGrid