# SPEAR Theorem Prover

Domagoj Babić *
(Theorem prover architect)
University of British Columbia

Frank Hutter
(Search parameter optimization)
University of British Columbia

## Abstract

SPEAR is a modular arithmetic theorem prover designed for proving software verification conditions. The core of the theorem prover is a fast and simple SAT solver, which is described in this paper.

***Keywords*** Theorem proving, boolean satisfiability, parameter optimization, modular arithmetic

## 1. Introduction

SPEAR is a theorem prover for modular arithmetic, designed for software verification, but is also fast on other industrial problems, like bounded hardware modelchecking. When given modular arithmetic constraints, SPEAR performs elaborate encoding and optimization of constraints. Together with structural information, the encoded formula is passed to the core SAT solver. Given CNF input, SPEAR acts like an ordinary SAT solver, and does not attempt to reconstruct structural information, which is typically lost when the industrial instances are encoded into CNF.

Three versions of the solver were submitted to the SAT 2007 competition:

- SPEAR v0.8 — Search parameters were manually optimized according to a "this should work" heuristic. Expensive simplifications, like variable and clause elimination, are disabled.
- SPEAR v0.8 FH — Search parameters were found by Frank Hutter using ParamILS [4], an automatic tool for parameter optimization based on iterated local search in configuration space. Expensive simplifications are disabled as well.
- SPEAR v0.8 FHS — Manual modification of the FH set of parameters so as to include expensive simplifications.

The following sections describe the main features of SPEAR architecture, and parameter optimization.

## 2. Architecture

The core of SPEAR is a simple lightweight DPLL SAT solver with highly optimized boolean constraint propagation (BCP), very similar to the BCP routine in HYPERSAT [1]. Several other features were borrowed from HYPERSAT: phase selection heuristic and algorithm for finding the next watched literal. Clause representation is similar as well. Several other features were modelled after Minisat [3]: frequent restarts and learned clause minimization. The implementation of the clause minimization was improved in several ways. For instance, Minisat uses stack-based work queue for clause minimization, while SPEAR uses FIFO, which has a much more predictable memory access pattern, and is easier to optimize.

Although the phase selection heuristic has been considered irrelevant, we found that phase selection can have significant effect on overall performance. A simple heuristic that always picks false phase first for each decision literal tends to perform well on instances generated from circuits. However, we found that the HYPERSAT phase selection heuristic performs much better in general. Depending on the average length of implication chains, HYPERSAT picks either the phase with more or less enqueued clauses on watched lists. If implication chains are long, implying the phase that results in more unsatisfied clauses increases the likelihood of running into a conflict, effectively decreasing the average length of implication chains. If the chains are short (more frequent case for industrial benchmarks), picking the phase that satisfies more watched clauses tends to reduce the total amount of computation. Since the second case is more frequent, that is the default phase selection heuristic in SPEAR.

SPEAR is very configurable. Almost all search parameters (roughly 25 parameters) are modifiable from the command line. Besides setting individual parameters, SPEAR also supports predefined parameter sets for specific problems. This is an important feature, because various combinations of parameters can have drastic effects on the runtimes. For instance, even with very lightweight parameter optimization over a diverse set of instances, we observed $> 56$ X performance improvement on software verification instances. With default parameters, 287 software verification instances were solved in $160,441$ sec, with 38 timeouts, while with the optimized parameters (FH version), the same set of instances is solved in $2,857$ sec, without timeouts. This large improvement was achieved without optimizing the parameters specifically for software verification problems, so we expect even more significant speedup once we optimize the parameters specifically for software verification problems. The next section presents parameter optimization in more detail.

## 3. Parameter Optimization

Determining appropriate values for an algorithm's free parameters is a challenging and cumbersome task in the design of effective algorithms for hard problems. It is, however, well worth the effort since good parameter settings often make the difference between solving a problem in seconds and solving it in hours (or not at all).

We argue that for complex parameter tuning tasks automatic (or semi-automatic) approaches can outperform manual approaches while at the same time considerably reducing the time algorithm designers need to spend for tuning their algorithms. During development, algorithm designers typically only track performance on a few instances, limiting expensive batch experiments to infrequent intervals. This bears the risk of "over-tuning" performance to the used instances with poor generalization to other instances, even ones with very similar characteristics [2, 4]. Further, humans tend to focus on single algorithm components instead of grasping the complex interplay of all components taken together.

* His research is supported by a Microsoft Graduate Fellowship.

Automatic tools for parameter optimization also pave the way to an automatic algorithm design, viewed as the combination of alternative building blocks. For example, two tree search algorithms that only differ in their preprocessing and variable heuristics can be seen as a single algorithm with two nominal parameters. Thus, constructing the best algorithm for a domain can be seen as a parameter optimization problem.

SPEAR is an excellent testbed for automatic parameter optimization for the following reasons:

- It has a large number of parameters of various types. Its 25 parameters include categorical choices between heuristics, nominal parameters, as well as integer and continuous parameters.
- It shows state-of-the-art performance for a practically relevant class of problem instances, and tuning it will thus be of high practical relevance. In particular, in our experimental analysis SPEAR consistently showed the best results for solving software verification instances.

The second author is currently performing research in automatic methods for parameter optimization (for both local search and tree search algorithms) and we used SPEAR for a case study in parameter optimization. The algorithm designer (first author) provided a binary of SPEAR and information about its parameters and loose sensible values for each of them; the default parameter setting, however, was not revealed. The goal of this study was to see whether the performance achieved with automatic methods could rival the performance achieved by the manually engineered default parameters.

The particular method used for parameter optimization is called ParamILS and views parameter tuning as an optimisation problem [4]. In a nutshell, it performs an iterated local search in parameter configuration space, computing the objective function to be maximized as the geometric mean speedup over the default parameters. Since the optimization objective was good performance for industrial instances in the SAT competition 2007, we used the following instances for training and evaluation: the 176 industrial instances from the SAT competition 2005, the 200 instances from the SAT Race 2006, as well as 30 software verification instances generated by the first author. 300 of these 404 instances were used for training, the remaining 104 test instances only being used to get an unbiased performance estimate of our final tuned parameter setting.

During training, we took the risk of setting a low timeout of 10 seconds in order to save time. This bore the possibility of over-tuning the solver for good performance on short runs but poor performance on longer runs, and the domain expert (the first author) was indeed worried that the parameter setting would be too aggressive, leading to poor performance on harder instances. However, our experimental results do not support this fear.

Figure 1 compares the performance of our automatically found parameter setting against the manually engineered default, using $1,000$ seconds as a timeout for each of the 404 instances. The default timed out on 96 instances, the tuned one on 85 (74 instances remained unsolved by either approach). For the remaining points, the tuned parameter setting achieved a geometric mean speedup of 21%, with a trend to perform better for larger instances (reducing our worries about over-tuning to easy instances). In the figure, we distinguish training and test instances in order to test whether performance on the training instances would be much better. Clearly, empirical results show no evidence of overfitting.

Overall, this result demonstrates that an automatic tuning approach can indeed outperform manually engineered parameter settings. Performance speedups were especially large for software verification instances: with an independent test set of 287 instances,

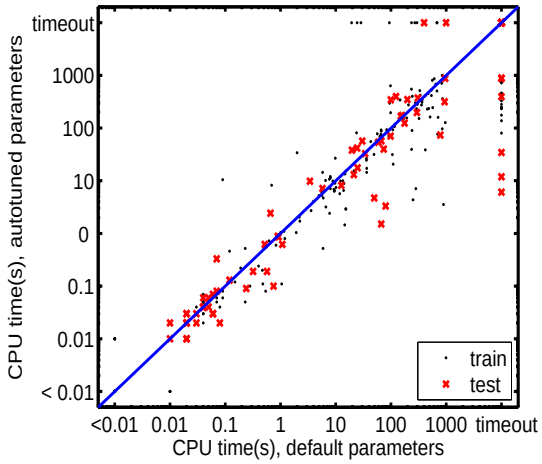


**Figure 1.** Performance of tuned SPEAR parameters vs. its defaults on training and test data.

the first author found a more than 56-fold speedup of the tuned parameter settings over the default.

Finally, parameter setting SPEAR V0.8 FHS is a variation of our tuned parameter setting SPEAR V0.8 FH that also includes expensive simplifications (which had not been implemented at tuning time). The simplification parameters in SPEAR V0.8 FHS were manually tuned on a few instances, and we cannot say anything about their generalization performance. In the future, we anticipate to speed up our optimization techniques, such that an interleaved parameter tuning may become possible after each modification of the code base.

## 4. Future Work

In the near future, SPEAR will support Satisfiability Modulo Theories (SMT) modular arithmetic format with additional optimizations. The SAT solver will also get more structural information about the instance being solved, which, we hope, will result in even better performance.

On the parameter optimization side, we plan to optimize the solver for several important industrial classes of problems (like model checking and software verification) and offer more predefined sets of options for those specific classes of problems.

## References

[1] Domagoj Babic, Jesse Bingham, and Alan J. Hu. B-cubing: New possibilities for efficient sat-solving. *IEEE Trans. Comput.*, 55(11):1315–1324, 2006.

[2] M. Birattari. *The Problem of Tuning Metaheuristics as seen from a Machine Learning perspective*. PhD thesis, Universite Libre de Bruxelles, Facult'e des Sciences Appliqu'ees, IRIDIA, Institut de Recherches Interdisciplinaires et de D'eveloppements en Intelligence Artificielle, 2005.

[3] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[4] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. Under review.