Boosting Verification by Automatic Tuning of Decision Procedures

Frank Hutter, Domagoj Babić, Holger H. Hoos, and Alan J. Hu
Department of Computer Science, University of British Columbia
{hutter, babic, hoos, ajh}@cs.ubc.ca

***Abstract*—Parameterized heuristics abound in computer aided design and verification, and manual tuning of the respective parameters is difficult and time-consuming. Very recent results from the artificial intelligence (AI) community suggest that this tuning process can be automated, and that doing so can lead to significant performance improvements; furthermore, automated parameter optimization can provide valuable guidance during the development of heuristic algorithms. In this paper, we study how such an AI approach can improve a state-of-the-art SAT solver for large, real-world bounded model-checking and software verification instances. The resulting, automatically-derived parameter settings yielded runtimes on average 4.5 times faster on bounded model checking instances and 500 times faster on software verification problems than extensive hand-tuning of the decision procedure. Furthermore, the availability of automatic tuning influenced the design of the solver, and the automatically-derived parameter settings provided a deeper insight into the properties of problem instances.**

***Index Terms*—Decision Procedures, Boolean Satisfiability, Search Parameter Optimization**

I. Introduction

The problems encountered in automated formal verification are typically hard. As with other computationally difficult problems, the key to practical solutions lies in the use of heuristic techniques. In the context of verification, decision procedures, which might be embodied as a BDD [1] package, a Boolean satisfiability (SAT) solver (e.g., [2]), or an automated theorem prover based on the Nelson-Oppen framework [3], all make use of various heuristics that have a crucial impact on their performance.

A high-performance decision procedure typically uses multiple heuristics that interact in complex ways. Some examples from the SAT-solving world include decision variable and phase selection, clause deletion, next watched literal selection, and initial variable ordering heuristics (e.g., [4], [5], [6]). The behavior and performance of these heuristics is typically controlled by parameters, and the complex effects and interactions between these parameters render their tuning extremely challenging.

During the typical development process of a heuristic solver, certain heuristic choices and parameter settings are tested incrementally, typically using a modest collection of benchmark instances that are of particular interest to the developer. Many choices and parameter settings thus made are "locked in" during early stages of the process, and typically, only few parameters are exposed to the users of the finished solver. In many cases, these users never change the default settings of the exposed parameters or manually tune them in a manner similar to that used earlier by the developer.

Not surprisingly, this manual configuration and tuning approach typically fails to realize the full performance potential of a heuristic solver. In this paper, we present an alternative approach based on automated parameter optimization methods and demonstrate its benefits, which include substantial performance improvements, valuable guidance to the algorithm designer, and new insights into specific types of (SAT-encoded) verification problems.

Specifically, we explain how PARAMILS, a recent parameter optimization tool developed by Hutter et al. [7], was used during the development of SPEAR, a high-performance modular arithmetic decision procedure and SAT solver, which was developed in support of the CALYSTO static checker [8]. Although the performance of an early, manually-tuned version of SPEAR was comparable to that of a state-of-the-art SAT solver (MiniSAT 2.0 [9]), the use of PARAMILS ultimately lead to speedups between a factor of 4.5 and a factor of 500 due to the optimization of the search parameters. The use of PARAMILS also influenced the design of SPEAR and gave us some important insights about differences between (SAT-encoded) hardware and software verification problems; for example, we found that the software verification instances generated by the CALYSTO static checker required more aggressive use of SPEAR's restart mechanism than the bounded model checking hardware verification benchmarks we studied.

While the results of our case study are interesting in their own right, it should be noted that our overall approach and the specific parameter optimization tool used in this study are very general and can be applied to any parameterized heuristic algorithm; the performance criterion that is automatically optimized can be runtime, precision, latency, or any other computable scalar metric.

II. Related Work

There are almost no publications on automated parameter optimization for decision procedures for formal verification. Seshia [10] explored using support vector machine (SVM) classification to choose between two encodings of difference logic into Boolean SAT. The learned classifier was able to choose the better encoding in most instances he tested, resulting in a hybrid encoding that mostly dominated the two pure encodings. The only other work we are aware of is unpublished, ad hoc work in industry.

There is, however, a fair amount of previous work on optimizing SAT solvers for particular applications. For example, Shtrichman [11] considered the influence of variable and phase decision heuristics (especially static ordering), restriction of the set of variables for case splitting, and symmetric replication of conflict clauses on solving bounded model checking (BMC) problems. He evaluated seven strategies on the Grasp SAT solver, and found that static ordering does perform fairly well, although no parameter combination was a clear winner. Later, Shacham and Zarpas [12] showed that Shtrichman's conclusions do not apply to zChaff's less greedy VSIDS heuristic on their set of benchmarks, claiming that Shtrichman's conclusions were either benchmark- or engine-dependent. Shacham and Zarpas evaluated four different decision strategies on IBM BMC instances, and found that static ordering performs worse than VSIDS-based strategies. Lu at al. [13] exploited signal correlations to design a number of ATPG-specific techniques for SAT solving. Their technique showed roughly an order of magnitude improvement on a small set of ATPG benchmarks.

The automated parameter optimization tool used in our study has been recently introduced by Hutter et al. [7]; however, that work was more focused on theoretical properties of the algorithm and did not consider an application to a state-of-the-art solver for real-world problems. That work and the study presented here complement each other and also address two different communities. Very broadly, automated parameter optimization can be seen as as a stochastic optimization problem that can be solved using a range of generic and specific methods [14], [15], [16]. However, these are either limited to algorithms with continuous parameters or algorithms with a small number of discrete parameters.

III. Algorithm Development and Manual Tuning

The core of Spear is a DPLL-style [17] SAT solver, but with several novelties. For example, Spear features an elaborate clause prefetching mechanism that improves memory locality. To improve the prediction rate of the prefetching mechanism, Boolean constraint propagation (BCP) and conflict analysis have been redesigned to be more predictable. Spear also features novel heuristics for decision making, phase selection, clause deletion, and variable and clause elimination. In addition, Spear has several enhancements for software verification, such as support for modular arithmetic constraints [18], incrementality to enable structural abstraction/refinement [8], and a technique for identifying context-insensitive invariants to speed up solving multiple queries that share common structure [19]. Given all of these features, extensions, and heuristics, many components of Spear are parameterized, including the choice of heuristics, as well as enabling (or disabling) of various features: e.g., pure-literal rule, randomization, clause deletion, and literal sorting in freshly learned clauses. Thus, the optimization of these parameters is a challenging task.

After the first version of Spear was written and its correctness thoroughly tested, its developer, Domagoj Babić, spent one week on manual performance optimization, which involved: (i) optimization of the implementation, resulting in a speedup by roughly a constant factor, with no effects on the search parameters, and (ii) manual optimization of roughly twenty search parameters, most of which were hard-coded and scattered around the code at the time.

The manual parameter optimization was a slow and tedious process done in the following manner: the Spear developer collected several medium-sized benchmark instances which it could solve in at most 1000 seconds and attempted to come up with a parameter configuration that would result in a minimum total runtime on this set. The benchmark set was very limited and included several medium-sized BMC and some small software verification (SWV) instances generated by the Calysto static checker [8].[1] Such a small set of test instances facilitates fast development cycles and experimentation, but has many disadvantages.

Quickly it became clear that implementation optimization gave more consistent speedups than parameter optimization. Even on such a small set of benchmarks, the variations due to different parameter settings were huge. We even found one case (Alloy analyzer [20] instance handshake.als.3) where the difference of floating point rounding errors between Intel's non-standard 80-bit and IEEE 64-bit precision resulted in an extremely large difference in the runtimes on the same processor. The same instance was solved in 0.34 sec with 80-bit precision and timed out after 6000 sec with 64-bit precision. The difference in rounding initially caused minor differences in variable activities, which are used to compute the dynamic decision ordering. Those minor differences quickly diverged, pushing the solver into two completely different parts of search space. Since most parameters influence the decision heuristics in some way, the solver might be equally sensitive to parameter changes.[2]

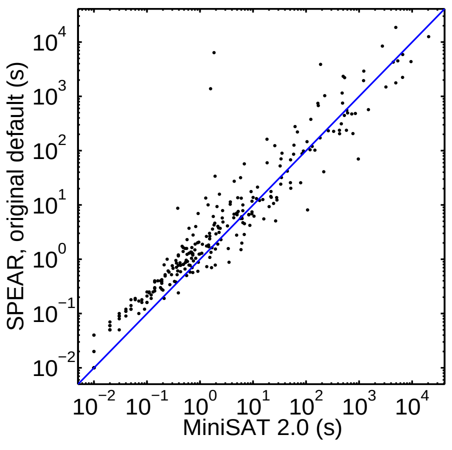
Given the costly and tedious nature of the process, no further manual parameter optimization was performed after finding a configuration that seemed to work well on the chosen test set.

To assess the performance of this manually tuned version of Spear, we ran it against MiniSAT 2.0 [9], the winner of the industrial category of the 2005 SAT Competition and of the 2006 SAT Race. In this experiment, we used two instance sets introduced in detail later in Sec. V: bounded model checking (BMC) and software verification (SWV). As can be seen from the runtime correlation plots shown in Figure 1, both solvers perform quite similarly for bounded model checking and easy software verification instances. For difficult software verification instances, however, MiniSAT clearly performs better. This seems to be the effect of focusing the manual tuning on a small number of easy instances.

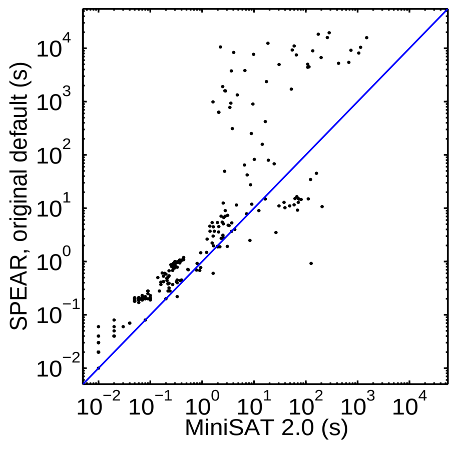
For most decision procedures, the process of finding default (or hard-coded) parameter settings resembles the manual tuning described above. Furthermore, most users of these tools

[1]Small instances were selected because Calysto tends to occasionally generate very hard instances that would not be solved within a reasonable amount of time.

[2]This emphasizes the need to find parameter settings that lead to more robust performance, with different random seeds, as well as across instances.



(a) Bounded model checking



(b) Software verification

Fig. 1. MiniSAT 2.0 vs. SPEAR using its original, manually tuned default parameter settings. (a) The two solvers perform comparably on bounded model checking instances, with average runtimes of 298 seconds (MiniSAT) vs. 341 seconds (SPEAR) for the instances solved by both algorithms. (b) Performance on easy and medium software verification instances is comparable, but MiniSAT scales better for harder instances. The average runtimes for instances solved by both algorithms are 30 seconds (MiniSAT) and 787 seconds (SPEAR).

do not change these settings, and when they do, they typically apply the same manual approach.

IV. Parameter Optimization by Local Search

The tool we chose to use for automatically optimizing parameter settings in SPEAR has recently been developed in the Artificial Intelligence community [7]; in the following, we briefly introduce the underlying PARAMILS algorithm (further details and some theoretical background can be found in the paper by Hutter et al. [7]).

PARAMILS is motivated by the following manual parameter tuning technique often used by algorithm developers:

- Start with some parameter configuration
- Iteratively, modify one algorithm parameter at a time, keeping the modification if performance on a given benchmark set improves and undoing it otherwise.
- Terminate when no single parameter modification yields an improvement, or when the best configuration found so far is considered "good enough".

Notice that this is essentially a simple hill-climbing local search process, and as such it will typically terminate in a locally, but not globally optimal parameter configuration, in which changing any single parameter value will not achieve any performance improvement. However, since parameters of heuristic algorithms are typically not independent, changing two or more parameter values at the same time may still improve performance.

The problem of local optima is ubiquitous in local search, and many approaches have been developed to effectively deal with them; one of these approaches is *Iterated Local Search (ILS)* [21], [22], which provides the basis for PARAMILS. ILS essentially alternates a subsidiary local search procedure (such as simple hill-climbing) with a perturbation phase, which lets the search escape from a local minimum. Additionally, an acceptance criterion is used to decide whether to continue the search from the most recently discovered local minimum or from some earlier local minimum. More precisely, starting from some initial parameter configuration, PARAMILS first performs simple hill-climbing search until a local minimum c is reached, and then it cycles through the following phases:

1) apply perturbation (in the form of multiple random parameter changes);
2) perform simple hill-climbing search until a new local minimum c' is reached;
3) accept the better of the two configurations c and c' as the starting point of the next cycle.

PARAMILS thus performs a biased random walk over locally optimal parameter configurations. To determine the better of two configurations, it can use arbitrary scalar performance metrics, including expected runtime, expected solution quality (for optimization algorithms), or any other statistic on the performance of the algorithm to be tuned when applied to instances from a given benchmark set. This benchmark set is called the *training set*, in contrast to the *test sets* we used later for evaluating the final parameter configurations obtained from PARAMILS (as is customary in the empirical evaluation of machine learning algorithms, training and test sets are strictly disjoint).

Clearly, the choice of the training set has important consequences for the performance of PARAMILS. Ideally, a homogenous training set would be chosen, i.e., one in which the impact of parameter settings on the performance of the algorithm to be tuned (here, SPEAR) is similar for all instances in the set. In that case, it would be sufficient and 'safe' to evaluate and compare parameter configurations by running the solver on a small number of instances. In practice, however, 'interesting' instance sets may not be homogenous, and therefore larger training sets may be required to achieve a reasonably unbiased evaluation of parameter configurations.

BASICILS(N) is a simple version of PARAMILS that uses a training set of N instances, where the choice of N has a major impact on the efficacy of the tuning process. For small N, there is a risk of over-fitting, i.e., good parameter configurations determined for the corresponding small sets may be overly specific to the training set and not work well for any other problem instances. For large N, however, the evaluation of each parameter configuration becomes costly, which can severely limit the number of search steps that can be practically performed by PARAMILS (and hence reduce the quality of the final parameter configuration returned by the tuning algorithm).

FOCUSEDILS is a more advanced version of PARAMILS. It adaptively chooses the number of training instances to use for each parameter setting: while poor settings can be discarded after a few algorithm runs, promising ones are evaluated on more instances. This mechanism avoids over-fitting to the instances in the training set. (For details, see [7].) In tuning SPEAR, we initially used BASICILS(300) and later employed the more advanced FOCUSEDILS.

V. AUTOMATED PARAMETER OPTIMIZATION

We performed two sets of experiments: automated tuning of SPEAR on a general set of instances for the 2007 SAT competition and application-specific tuning for two real-world benchmark sets.

A. Benchmark Sets and Experimental Setup

We employed two sets of problems of immense practical importance: hardware bounded model checking and software verification. Specifically, our set of BMC instances consists of 754 IBM BMC instances created by Zarpas [23], and our SVW benchmark set is comprised of 604 verification conditions generated by the CALYSTO static checker [8].

Both instance sets, BMC and SWV, were split 50:50 into disjoint training and test sets. Only the training sets were used for tuning, and all results in this paper are for the test sets. All reported experiments were carried out on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 10.1. Reported times are CPU times per single CPU. Runs are terminated after 10 CPU hours or when they run out of memory and start swapping; we count both of these conditions as time-outs.

B. Search Parameters

The availability of automatic parameter tuning encouraged us to parameterize many aspects of SPEAR. The first automatically tuned version exposed only a few important parameters, such as restart frequencies and variable priority increments. The results of automated tuning of those first versions of SPEAR prompted its developer to expose more and more search parameters, up to the point where not only every single hard-coded parameter was exposed, but also a number of new parameter-dependent features were incorporated. This process not only significantly improved SPEAR's performance, but also has driven the development of SPEAR itself.

The resulting version of SPEAR used for the experiments reported in the following has 26 parameters:

- 7 types of heuristics (with the number of different heuristics available shown in parentheses):
  - Variable decision heuristics (20)
  - Heuristics for sorting learned clauses (20)
  - Heuristics for sorting original clauses (20)
  - Resolution ordering heuristics (20)
  - Phase selection heuristics (7)
  - Clause deletion heuristics (3)
  - Resolution heuristics (3)
- 12 double-precision floating point parameters, including variable and clause decay, restart increment, variable and clause activity increment, percentage of random variable and phase decisions, heating/cooling factors for the percentage of random choices, etc.
- 4 integer parameters which mostly control restarts and variable/clause elimination.
- 3 Boolean parameters which enable/disable simple optimizations such as the pure literal rule.

For each of SPEAR's floating point and integer parameters we chose lower and upper bounds on reasonable values and considered a number of values spread uniformly across the respective interval. This number ranges from three to eight, depending on our intuition about importance of the parameter. The total number of possible combinations after this discretization is 3.78×10^{18}. By exploiting some dependencies between parameters, we reduced the number of configurations that we consider in this paper to 8.34×10^{17}.

C. SAT Competition Tuning

The first round of automatic parameter optimization was done in the context of preparing a version of SPEAR for submission to the 2007 SAT Competition. The first two authors used this as a case study in parameter optimization for real-world problem domains: the SPEAR developer provided an executable of SPEAR and information about its parameters as well as approximate ranges of reasonable values for each of them; the default parameter configuration, 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.

Since the optimization objective was to achieve good performance on the industrial benchmarks of the 2007 SAT Competition (which were not disclosed before the solver submission deadline), we used a collection of instances from previous competitions for tuning: 176 industrial instances from the 2005 SAT Competition, 200 instances from the 2006 SAT Race, as well as 30 SWV instances generated by the CALYSTO static checker. A subset of 300 randomly selected instances was used for training, and the remaining 106 test instances provided an unbiased performance estimate of SPEAR's performance with the tuned parameter configuration. Since the SAT competition rules reward per-instance performance relative to other solvers, the optimization objective used in this phase was geometric mean speedup over SPEAR with the (manually optimized) default parameter settings.

We ran a single run of BASICILS(300) for three days on the 300 designated training instances, and used the parameter configuration with the best training set performance found within that time; we refer to this parameter configuration as `Satcomp`. During tuning, we took the risk of setting a low cutoff time of 10 seconds for each single algorithm run 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 we expected that parameter configuration `Satcomp` may be too aggressive and might perform poorly on harder instances.

However, our experimental results indicate that the opposite is the case, namely that SPEAR's performance scales better with the `Satcomp` parameter settings than with the default settings. The fact that these results contradicted the intuition of the algorithm's developer illustrates clearly the limitations of even an expert's ability to comprehend the complex interplay between the many parameters of a sophisticated heuristic algorithm such as SPEAR.

On the 106 test instances used to assess the result of our SAT competition tuning, `Satcomp` achieved a geometric mean speedup of 21% over SPEAR's default parameter settings and showed much better scaling with instance hardness. Figure 2 demonstrates that this speedup carries over to both our verification benchmark sets: `Satcomp` performs better than the SPEAR default on BMC (with an average speedup factor of about two) and clearly dominates it for SWV (with an average speedup factor of about 78).

D. Application-specific Tuning

While general tuning on a mixed set of instances as performed for the 2007 SAT Competition resulted in a solver with strong overall performance, in practice, one often mostly cares about excellent performance on a specific type of instances, such as BMC or SWV. For this reason we performed a second set of experiments — tuning SPEAR for these two specific sets of problems. Since users typically care most about an algorithm's total runtime, we used average (arithmetic mean) runtime as our optimization objective in this tuning phase.

For both sets, during training we chose a cutoff of 300 seconds, which according to SPEAR's internal book-keeping mechanisms turned out to be sufficient for exercising all techniques implemented in the solver. In order to speed up the optimization, in the case of BMC we removed 95 hard instances from the training set that could not be solved by SPEAR with its default parameter configuration within one hour, leaving 287 instances for training.

We performed parameter optimization by running 10 parallel copies of FOCUSEDILS on a cluster, for three days in the case of SWV and for two days for BMC. For each instance set, we picked the parameter configuration with the best training performance after that time.

Figure 3 demonstrates that these application-specific parameter configurations perform even better than the optimized settings for the SAT competition, `Satcomp`. SPEAR's performance is boosted for both application domains, by an average factor of over 2 for BMC and over 20 for SWV; the scaling behavior also clearly improves, especially for SWV.

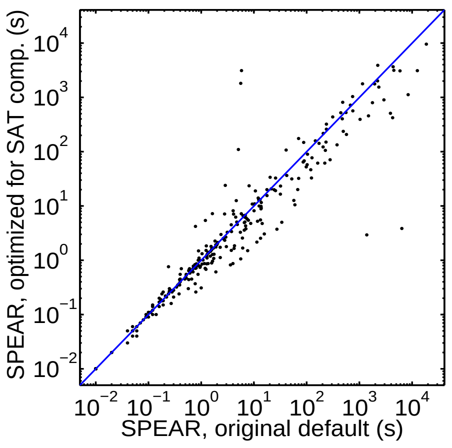
Figure 4 shows the total effect of automatic tuning by comparing the performance of SPEAR with the (manually optimized) default settings against that achieved when using the parameter configurations tuned parameters for the BMC and SWV benchmark sets. For both sets, the scaling behavior of the tuned version is much better and on average, large speedups are achieved — by a factor of 4.5 for BMC and 500 for SWC. SPEAR with the default settings even times out on four SWV instances after 10 000 seconds, while the tuned version solves every single instance in less than 20 seconds.

Figure 5 summarizes the performance of MiniSAT 2.0 (which we used as a baseline) and SPEAR with parameter settings default, `Satcomp`, and specifically tuned for BMC and SWV. Notice that the versions of SPEAR specifically tuned for BMC and SWV also clearly outperform MiniSAT: for BMC, SPEAR solves two additional instances and is faster by a factor of three on average; for SWV, the speedup factor is over 100. For both benchmark sets, scatter plots (not shown here) also reveal much better scaling behavior of the specifically tuned versions of SPEAR.

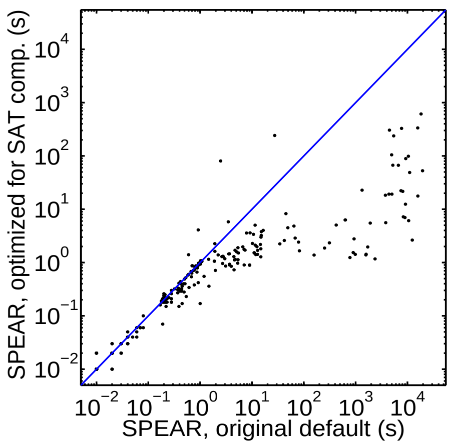
VI. DISCUSSION

Automated parameter tuning provided us with new insights into properties of the benchmark instances used in our study and influenced the design of SPEAR. These insights arise from considering characteristic differences between the optimized parameter configurations for the BMC and SVW instances.

Although we have limited knowledge about the high-level features of the IBM BMC instances, we made some interesting observations. The best decision heuristic that we found for these instances picks variables with higher activity, and ties are resolved by choosing the one with a smaller product of positive and negative occurrences. We also found that the IBM BMC instances favor less aggressive restarts than the SVW instances, implying that the decision heuristic tends to find better variable orderings. The best phase selection heuristic we found for BMC instances aggressively picks the phase so as to minimize the number of watched clauses that need to be traversed in order to find the next watched literal. This heuristic minimizes the number of clauses that BCP needs to analyze, and its effectiveness on this hard set of instances did not surprise us. Finally, we observed that a small amount of randomness helps performance — roughly 5% of phase and variable decisions were done randomly before the first restart. The most effective strategy scales down the percentage

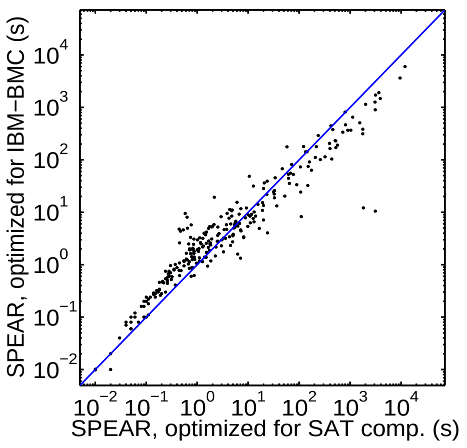


(a) Bounded model checking

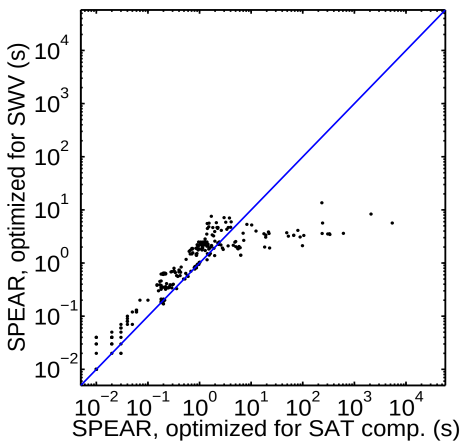


(b) Software verification

Fig. 2. Improvements by automated parameter optimization on a mix of industrial instances: SPEAR with the original default parameter configuration vs. SPEAR with configuration `Satcomp`. (a) Even though a few instances can be solved faster with the SPEAR default, parameter configuration `Satcomp` is considerably faster on average (mean runtime 341 vs. 223 seconds). Note that speedups are larger than they may appear in the log-log plot: for the bulk of the instances `Satcomp` is about twice as fast. (b) `Satcomp` improves much on the scaling behavior of the SPEAR default, which fails to solve four instances in 10 000 seconds. Mean runtimes on the remaining instances are 787 seconds vs. 10 seconds, a speedup factor of 78.

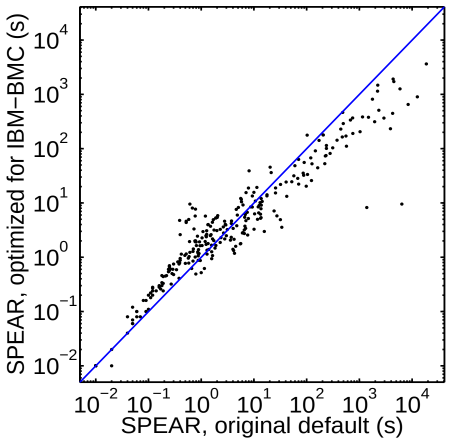


(a) Bounded model checking

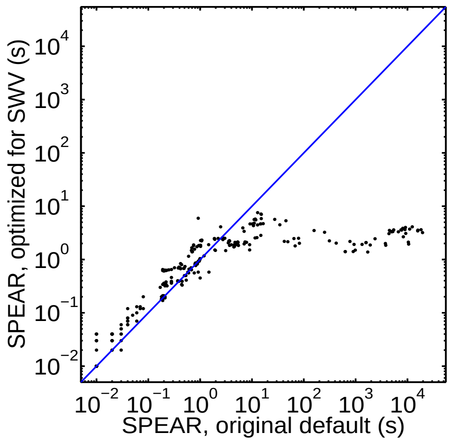


(b) Software verification

Fig. 3. Improvements by automated parameter optimization on specific instance distributions: SPEAR with configuration `Satcomp` vs. SPEAR with parameters optimized for the specific applications BMC and SWV. Results are on independent test sets disjoint from the instances used for parameter optimization. (a) The parameter configuration tuned for set BMC solved four instances for which configuration `Satcomp` timed out after 10 000 seconds. For the remaining instances, mean runtimes are 223 seconds (`Satcomp`) and 96 seconds (specific tuning for BMC), a speedup by more than a factor of two. (b) Both parameter settings solved all 302 instances, mean runtimes are 36 seconds (`Satcomp`) and 1.5 seconds (tuned for SWV), a speedup factor of 24.



(a) Bounded model checking



(b) Software verification

Fig. 4. Overall improvements achieved by automatic tuning: SPEAR with its manually engineered default parameter configuration vs. the optimized versions for sets BMC and SWV. Results are on test sets disjoint from the instances used for parameter optimization. (a) The default timed out on 90 instances after 10 000 seconds, while the tuned configuration solved four additional instances. For the instances that the default solved, mean runtimes are 341 seconds (default) and 75 seconds (tuned), a speedup factor of 4.5. (b) The default timed out on four instances after 10 000 seconds, the tuned configuration solved all instances in less then 20 seconds. For the instances that the default solved, mean runtimes are 787 seconds (default) and 1.35 seconds (tuned), a speedup factor of over 500.

Solver	Bounded model checking		Software verification	
	#(solved)	runtime for solved	#(solved)	runtime for solved
MiniSAT 2.0	289/377	360.9	302/302	161.3
SPEAR original	287/377	340.8	298/302	787.1
SPEAR general tuned	287/377	223.4	302/302	35.9
SPEAR specific tuned	291/377	113.7	302/302	1.5

Fig. 5. Summary of Results. For each solver and instance set, #(solved) denotes the number of instances solved within a CPU time of 10 hours, and the runtimes are the arithmetic mean runtimes for the instances solved by that solver. (Geometric means were not meaningful here, as all solvers solved a number of easy instances in "0 seconds"; arithmetic means better reflect practical user experience as well.) If an algorithm solves more instances, the shown average runtimes include more, and typically harder, instances. Note that the averages in this table differ from the runtimes given in the captions of Figures 1-4, because averages are taken with respect to different instance sets: for each solver, this table takes averages over all instances solved by that solver, whereas the figure captions state averages over the instances solved by both solvers compared in the respective figure.

of random decisions by a factor of 0.7 at each restart (which resembles the idea of simulated annealing).

Since we are intimately familiar with the CALYSTO static checker, we are able to provide a deeper analysis for the software verification instances. CALYSTO performs aggressive common subexpression elimination, virtually eliminating all symmetries. It also propagates all constants. CALYSTO queries correspond to path- and context-sensitive verification conditions, which have deep and rich Boolean structure, with many expensive operations (like division and multiplication) sprinkled around. The queries can be represented at a high level as single-rooted acyclic graphs. Experimental results (see [8]) suggest that the probability of infeasibility of a single path starting from the root of the formula is proportional to the length of the path — the longer the path, the more likely it is that it is infeasible. This can be exploited by a SAT solver by focusing the search on the expressions that are closer to the root of the tree.

SWV instances prefer an activity-based heuristic that resolves ties by picking the variable with a larger product of occurrences. This heuristic might seem too aggressive, but helps the solver to focus on the most frequently used common subexpressions. It seems that a relatively small number of expressions play a crucial role in (dis)proving each verification condition, and this heuristic quickly narrows the search down to such expressions. The SWV instances favored very aggressive restarts (first after only 50 conflicts), which in combination with our experimental results shows that most such instances can be solved quickly if the right order of variables is found. A simple phase selection heuristic (always assign FALSE first) seems to work well for SWV, and also produces more natural bug traces (small values of variables in the satisfying assignments). The SWV instances correspond to NULL-pointer dereferencing checks, and this phase selection heuristic attempts to propagate NULL values first (all FALSE), which explains its effectiveness. SWV instances prefer no randomness at all, which is probably the result of joint development of CALYSTO and SPEAR as a highly optimized tool chain for software verification.

The use of automated parameter optimization also influenced the design of SPEAR in various ways. An early version of SPEAR featured a nascent implementation of clause and variable elimination. Prior to using automated tuning, these mechanisms did not consistently improve performance, and therefore, considering the complexity of finalizing their implementation, the SPEAR developer considered removing them. However, these elimination techniques turned out to be effective after parameter tuning found good heuristic settings to regulate the elimination process. Another feature that was considered for removal was the pure literal rule, which ended up being useful for BMC instances (but not for SWV). Similarly, manual optimization gave inconclusive results about randomness, but automated optimization found that a small amount of randomness actually does help SPEAR in solving BMC (but not SWV) instances.

VII. CONCLUSIONS

In this work, we have demonstrated that by using a general parameter optimization method, PARAMILS, which is based on the idea of iterated local search in parameter configuration space, major performance improvements of a high-performance SAT solver, SPEAR, can be achieved. We believe that the resulting optimized version of SPEAR represents a considerable improvement in the state of the art of solving decision problems from hardware and software verification using SAT-solvers. Tuning SPEAR on a general set of industrial instances from previous SAT competitions already resulted in large improvements when compared to SPEAR's manually optimized default parameter setting. The greatest improvements, however, were achieved when tuning was performed on a specific, relatively homogenous class of problem instances. Average runtimes were reduced by a factor of 4.5 for bounded model checking instances and a factor of over 500 for software verification instances (see Figure 4). It is worth noting that prior to applying our automated tuning approach, considerable time had been invested by its author to manually tune SPEAR. This indicates that automated parameter optimization can be considerably more effective than manual tuning, and that the use of automated tuning procedures such as PARAMILS not only frees the algorithm designer (and user) from the typically tedious and time-consuming manual tuning task, but also helps to better exploit the full performance potential of a highly parameterized heuristic solver.

Not too surprisingly, our experimental results suggest that optimized search parameters are benchmark-dependent — which highlights the advantages of automated parameter tuning over the conventional manual approach. Furthermore, parameter tuning is obviously engine-dependent, due to complex interactions between various mechanisms implemented in a typical decision procedure.

We also illustrated how the use of automated parameter optimization provided guidance in the development of SPEAR and in particular encouraged its developer to expose a large number of parameters that could then be optimized. We are convinced that similar benefits will arise when applying our general approach in the development of other heuristic algorithms. Finally, comparing specifically optimized parameter configurations, we gained some insights into which components of SPEAR were particularly effective on the hardware and software verification instances considered here.

In future work, we intend to further explore the role of local search and machine learning strategies that support algorithm design and engineering tasks. We believe that the tuning procedure can be further improved, for example, by combining ideas from our current local-search-based approach with concepts from racing procedures, or by incorporating techniques from experimental design. We also see significant potential in instance-specific tuning methods, which use machine learning techniques to find good parameter settings for a given problem instance [24], and in reactive tuning strategies, which adapt parameter settings while a solver is running (utilizing information gathered while trying to solve the given instance) [25]. Finally, considering that many other design and engineering tasks involve heuristic algorithms, we are convinced that the use of automated algorithm configuration and parameter optimization procedures can lead to similarly substantial performance improvements as demonstrated here and hope to collect further evidence for this claim in the near future.

REFERENCES

[1] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[2] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification." *STTT: International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, April 2005.

[3] G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, 1979.

[4] J. P. M. Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms," in *EPIA '99: Proc. of the 9th Portuguese Conference on Artificial Intelligence*, ser. LNCS, vol. 1695. Springer, 1999, pp. 62–74.

[5] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *DAC '01: Proc. of the 38th conference on Design automation*. ACM Press, 2001, pp. 530–535.

[6] A. Bhalla, I. Lynce, J. de Sousa, and J. Marques-Silva, "Heuristic backtracking algorithms for SAT," in *MTV '03: Proc. of the 4th International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions*, 2003, pp. 69–74.

[7] F. Hutter, H. H. Hoos, and T. Stützle, "Automatic algorithm configuration based on local search," in *AAAI '07: Proc. of the Twenty-Second Conference on Artifical Intelligence*, 2007, pp. 1152–1157.

[8] D. Babić and A. J. Hu, "Structural Abstraction of Software Verification Conditions," in *Computer Aided Verification: 19th International Conference, CAV 2007*, ser. LNCS, vol. 4590. Springer, 2007, pp. 366–378.

[9] N. Eén and N. Sörensson, "An extensible SAT solver," in *SAT '03: Proc. of the 6th International Conference on theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 2919. Springer, 2003, pp. 502–518.

[10] S. A. Seshia, "Adaptive eager boolean encoding for arithmetic reasoning in verification," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-05-134, May 2005.

[11] O. Shtrichman, "Tuning SAT checkers for bounded model checking," in *CAV '00: Proc. of the 12th International Conference on Computer Aided Verification*, ser. LNCS, vol. 1855. Springer, 2000, pp. 480–494.

[12] O. Shacham and E. Zarpas, "Tuning the VSIDS Decision Heuristic for Bounded Model Checking," in *Fourth International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV 2003), May 29-30, 2003, Hyatt Town Lake Hotel, Austin, Texas, USA*. IEEE Computer Society, 2003, pp. 75–79.

[13] F. Lu, L.-C. Wang, K.-T. T. Cheng, J. Moondanos, and Z. Hanna, "A signal correlation guided ATPG solver and its applications for solving difficult industrial cases," in *DAC '03: Proc. of the 40th conference on Design automation*. ACM Press, 2003, pp. 436–441.

[14] J. C. Spall, *Introduction to Stochastic Search and Optimization*, ser. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., 2003.

[15] B. Adenso-Diaz and M. Laguna, "Fine-tuning of algorithms using fractional experimental design and local search," *Operations Research*, vol. 54, no. 1, pp. 99–114, Jan–Feb 2006.

[16] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A racing algorithm for configuring metaheuristics," in *GECCO '02: Proc. of the Genetic and Evolutionary Computation Conference*, 2002, pp. 11–18.

[17] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[18] D. Babić and M. Musuvathi, "Modular Arithmetic Decision Procedure," Microsoft Research Redmond, Tech. Rep. TR-2005-114, 2005.

[19] D. Babić and A. J. Hu, "Exploiting Shared Structure in Software Verification Conditions," in submission.

[20] D. Jackson, "Automating first-order relational logic," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 6, pp. 130–139, 2000.

[21] H. R. Lourenço, O. C. Martin, and T. Stützle, "Iterated local search," in *Handbook of Metaheuristics*. Kluwer, 2002, pp. 321–353.

[22] H. H. Hoos and T. Stützle, *Stochastic Local Search - Foundations & Applications*. Morgan Kaufmann, 2005.

[23] E. Zarpas, "Benchmarking SAT Solvers for Bounded Model Checking," in *SAT '05: Proc. of the 8th International Conference on Theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 3569. Springer, 2005, pp. 340–354.

[24] F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown, "Performance prediction and automated tuning of randomized and parametric algorithms," in *CP '06: Proc. of the Twelfth International Conference on Principles and Practice of Constraint Programming*, 2006, pp. 213–228.

[25] R. Battiti and M. Brunato, *Approximation Algorithms and Metaheuristics*. CRC Press, 2007, ch. 21: Reactive Search: Machine Learning for Memory-based Heuristics, pp. 21–1 – 21–17.