Understanding Bit-vector Arithmetic in Z3

Veselin Mitev[1]

Supervisors: Soham Chakraborty[1]**, Dennis Sprokholt**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Veselin Mitev
Final project course: CSE3000 Research Project
Thesis committee: Soham Chakraborty, Dennis Sprokholt, Andy Zaidman

An electronic version of this paper can be found at https://repository.tudelft.nl/.

Satisfiability modulo theories solvers serve as the backbone of software verifiers, static analyzers, and proof assistants; the versatile bit-vector arithmetic theory is particularly important for these applications. As solvers continue to be developed, they become more capable but also more difficult to configure optimally. The widely-used Z3 prover offers a multitude of configurable techniques for solving bit-vector problems, however, the pace of development has not allowed for a comprehensive comparative analysis of them. We study the available bit-vector algorithms and parallelization methods, on established sets of problems. Through an experimental evaluation, we find that properly configuring Z3 for a specific use case can have significant effects on performance. We offer guidance to developers on how to go about that, which should help to make formal verification tools more efficient.

1. Introduction

Z3 is a satisfiability modulo theories (SMT) solver [1] developed at Microsoft[1]. An SMT problem[2] is defined by a set of variables and a set of formulas that act as constraints on those variables. This declarative style of problem representation, requires nothing more from the user, who in the case of SMT solvers is a programmer, usually automatically generating queries. Given a problem, the SMT solver tries to find an assignment for each of the variables, such that all constraints are satisfied. If the solver determines that no such assignment exists, it will report that the instance is unsatisfiable. Eq. 1 is an example of an SMT instance[3].

$$\begin{gathered} x, y \in \mathbb{Z} \\ x + 2y = 7 \\ (x+1) \bmod y = 0 \\ (x+y)^2 = x^y + y^{y^2} \end{gathered} \tag{1}$$

SMT solving deals with constraint satisfaction problems, which are solved with a combinatorial search. In that way, it is similar to constraint programming. The main difference is that SMT solving came about from the software verification and automated theorem proving communities, whereas constraint programming's main applications are algorithmics problems, such as scheduling, matching and assignment, resource allocation, etc.

SMT [2] can also be seen as an extension of the Boolean satisfiability problem (SAT); it involves not only boolean logic, but also arithmetic theories for non-linear integers and floating points, bit-vectors, strings, and others. It may include arrays, quantifiers, functions, and combinations of multiple arithmetic theories, as shown[4] in Eq. 2.

$$\begin{gathered} x \in \mathbb{Z}, y \in \mathbb{Q} \\ \forall z \in \mathbb{Z} : (z > 0 \to z + y > 0) \\ x^2 + \lfloor y \rfloor = 15 \vee (y \geq 3.5 \wedge x \bmod 2 = 0) \\ x \geq 0 \wedge y \leq 10.0 \end{gathered} \tag{2}$$

[1] Moura, L. de, and Bjørner, N. (2008) *Z3: An Efficient SMT Solver*

[1] The Z3 Theorem Prover GitHub Repository

[2] The words *problem*, *instance*, *query*, and *benchmark* are used interchangeably in literature.

[3] One satisfying assignment of Eq. 1 would be $x = 3, y = 2$.

[2] De Moura, L., and Bjørner, N. (2011) *Satisfiability Modulo Theories: Introduction and Applications*

[4] Eq. 2 is unsatisfiable. In it, the equations

$$\begin{gathered} x^2 + \lfloor y \rfloor = 15 \\ y \geq 3.5 \qquad x \bmod 2 = 0 \\ x \geq 0 \qquad y \leq 10.0 \end{gathered}$$

are all integer/rational number formulas, which serve as propositions in the boolean formulas:

$$\begin{gathered} x^2 + \lfloor y \rfloor = 15 \vee (y \geq 3.5 \wedge x \bmod 2 = 0) \\ x \geq 0 \wedge y \leq 10.0 \end{gathered}$$

This example mixes integer, rational number, and boolean arithmetic. In fact, it can be seen as one formula that is a conjunction of all the constraints (Eq. 3). This is essentially how SMT solvers view the queries they are given.

$$(\forall z \in \mathbb{Z} : (z > 0)(z^2 > 0)) \wedge (x^2 + \lfloor y \rfloor = 15 \vee (y \geq 3.5 \wedge x \bmod 2 = 0)) \wedge x \geq 0 \wedge y \leq 10.0 \tag{3}$$

The pride of many SMT solvers is just how many different theories, types of variables, and operations they support. That being said, most applications and evaluations of SMT solvers focus on specific subsets of theories. For example, the `AUFNIRA` logic, as defined in SMT-LIB [3], includes linear and non-linear mixed integer arithmetic, arrays, quantifiers and uninterpreted functions. A simpler, but very versatile logic is `QF_BV`, which supports quantifier-free[5] bit-vector and boolean formulas only.

Bit-vector arithmetic[6] is concerned with fixed-length lists of bits. We can apply bit-wise boolean operations (`and`, `or`, `xor`, `not`) to bit-vectors, but also bit-shifting, concatenation, and extraction of bits. In addition, bit-vectors can also be used to represent fixed-size integers if the corresponding operations — multiplication, division, subtraction, etc. — are supported (Figure 1) . That is particularly valuable, since modern-day computers and computer programs use fixed-length integers: `i32`, `long`, `unsigned int`. This means that many proofs, especially when it comes to software correctness [4], can be constructed in terms of bit-vector operations. What is more, bit-vectors are relevant in a wide variety of other fields and applications: software testing [5], genetics [6], optimization of factory plants [7], CAD designs [8], verification of hardware [9], smart contracts, neural networks [10], and others.

[3] Barrett, C., Fontaine, P., and Tinelli, C. (2016) *The Satisfiability Modulo Theories Library (SMT-LIB)*, SMT-LIB Logics

[5] `QF` logics do not include induction and quantifiers (e.g. $\forall$, or $\exists$), arrays, or uninterpreted functions (i.e. functions for which the solver has to find a valid definition).

[6] An example of an bit-vector query is:

$$x, y \in \mathrm{BV}[4]$$
$$(x + y = 15) \vee (x^y = 10)$$
$$x \& y = y$$

Here, x and y are bit-vectors of length (width) 4. Arithmetic operations like $+$, $-$, $\leq$, etc., are defined as operations on unsigned integers (unless specifically stated otherwise). $\&$ is referring a bit-wise and operation.

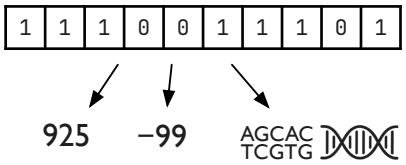


Figure 1. Bit-vectors can represent a variety of entities.

1.1. Contributions and Overview

With this paper, we answer the following questions about Z3 and its support for the bit-vector arithmetic theory.

1.1.1. What techniques can Z3 use to solve `QF_BV` problems?

Solvers use a variety of configurable algorithms and heuristics to efficiently solve problems, though not all of them are documented. We describe the broad architecture of Z3, the different bit-vector solving methods it supports, and the general idea behind how they work in Section 2.

[4] Cordeiro, L., Fischer, B., and Marques-Silva, J. (2009) *SMT-Based Bounded Model Checking for Embedded ANSI-C Software*

[5] Godefroid, P., Levin, M. Y., Molnar, D. A., et al. (2008) *Automated Whitebox Fuzz Testing.*

[6] Kroening, D., and Strichman, O. (2016) *Applications in Software Engineering and Computational Biology*, Section 12.4

In order to answer this question, we performed a literature review of papers about Z3, other SMT solvers, improvements to SMT solvers; we examined the documentation, codebase, release notes, commit history, issues and pull requests of Z3. Those sources have been cited at all the relevant places in this paper.

1.1.2. Which techniques work well with which sets of problems?

It is plausible that different techniques will be better suited to some sets of `QF_BV` problems than others. However, no comprehensive evaluation comparing all bit-vector algorithms in Z3 has been performed to determine whether that is the case.

In Section 4, we present our experimental evaluation of how well Z3's bit-vector algorithms perform on several known sets of benchmarks. We perform a statistical analysis of the results and, where applicable, compare them to other results from the literature in Section 4.3.1.

1.1.3. How is Z3 able to use parallelization when solving `QF_BV` problems?

Advancements in computing are trending towards having more computing units rather than individually faster ones [11]. CPU's have more and more cores, and even heterogeneous multicore architectures [12]. High-performance computing (HPC), distributed computing and GPU computing are also gaining traction [13, 14]. Thus, it is vital that advancements in software architectures try to make use of the available hardware support for concurrency.

Section 3 discusses what techniques Z3 uses to take advantage of modern-day multicore CPUs. Alongside that, we performed an experimental evaluation of how well Z3 scales with multicore architectures, which we describe in Section 4.3.2.

[11] Schauer, B. (2008) *Multicore Processors – A Necessity*

[12] Guertin, T., and Hurson, A. (2023) *The Multicore Architecture*

[13] Ramesh, R. S. (2024) *Scalable Systems and Software Architectures for High-Performance Computing on Cloud Platforms*

[14] Stoller, S., Carbin, M., Adve, S., et al. (2019) *Future Directions for Parallel and Distributed Computing*

1.2. Related Work

A very useful, though as-yet incomplete, account of the inner workings of Z3 was created by its developers [15]. It is an invaluable resource for anyone who wants to help in the development of Z3 or to just understand how state-of-the-art SMT solvers work and the details of some of the techniques that they employ. The document deals with a lot more than just bit-vectors, e.g., quantifiers, unsatisfiability

[15] Nikolaj Bjørner, Clemens Eisenhofer, Arie Gurfinkel, et al. *Z3 Internals (Draft)*

proofs, and other theories. It does omit a lot of details and explanations of more established techniques.

The Z3 Guide[7] is the main piece of documentation that anyone wanting to use Z3 should read. It introduces the basics of SMT solving, as well as the API of Z3 and the large amount of features it exposes. It it not fully up-to-date with all the capabilities of Z3 — there are some missing features, such as the int-blasting technique — but it is a good starting point.

[7] Z3 Guide powered by RiSE at Microsoft. Main contributors are Nikolaj Bjørner and Ruanqianqian (Lisa) Huang.

Related to the question posed in Section 1.1.2, Lu et al. [16] created `Z3alpha`. It is a very promising extension of Z3 that tries to automatically find good configurations. It utilizes pre-training and Monte Carlo tree search (MCTS) to synthesize a configuration tailored to each individual query. Their method offers an improvement of 42.7% more instances solved than the default Z3 configuration, within the time limit of a challenging benchmark suite. That being said, `Z3alpha` focuses on the preprocessing setup of Z3. It is not able to choose between the different bit-vector-specific algorithms that we identify in Section 2.3 — Section 2.6 and experiment with in Section 4.

[16] Lu, Z., Siemer, S., Jha, P., et al. (2024) *Layered and Staged Monte Carlo Tree Search for SMT Strategy Synthesis*

2. How Z3 Solves Bit-vector Problems

Z3 employs many of the same techniques that most other solvers use. However, one unique feature of Z3 is its configuration scheme, which allows users to make big-step strategic decisions about how the solver works.

2.1. Strategy and Tactics

The best SMT solvers, automated provers, and constraint programming engines all heavily rely on configurable heuristics to guide the design of their architectures, as well as the finer details of their implementations [17]. High-performance solvers make assumptions about the properties of the problems being solved. Using heuristics is necessary to achieve success on the rigorous benchmarks featured in SMT competitions. In fact, many of the advancements in constraint solving can actually be considered heuristics.

[17] Moura, L. de, and Passmore, G. O. (2013) *The Strategy Challenge in SMT Solving*, p. 17

Instead of relying on a fixed set of heuristics, Z3 allows its users to apply strategic control over how it solves problems given to it [17]. It does this via a modularized architecture based on so-called *tactics*, which are the building

blocks of the solver. Each tactic can either transform the problem into one or more sub-problems[8], or solve it outright[9]. For example, there are preprocessing tactics such as `simplify`, `ctx-simplify`, `lift-if`, `gaussian`, `aig`, `elim-predicates`, and solver tactics: `smt`, `nlsat`, and `qsat`.

Programmers can combine tactics in different ways via so-called `tacticals`. They take one or more tactics and produce another tactic that is, in some way, the combination of the two. Examples of `tacticals` include `then`, `try-for` (*a certain amount of time*), `par-or` (more about this in Section 3.2). This way, users can create pipleines of tactics that in essence define the big-picture architecture of how problems are solved. An example of a pipeline of tactics and how it solves a problem can be seen in Figure 2.

[8] Z3 calls the input and output of tactics goals. A tactic can produce multiple goals. In that case, solving any of the goals would produce a valid solution to the whole problem.

[9] Tactics can also fail. For instance, some tactics may only be tailored to work with certain classes of problems, or certain arithmetic theories. If such tactics detect anything outside of what they are capable of solving, they will simply fail.

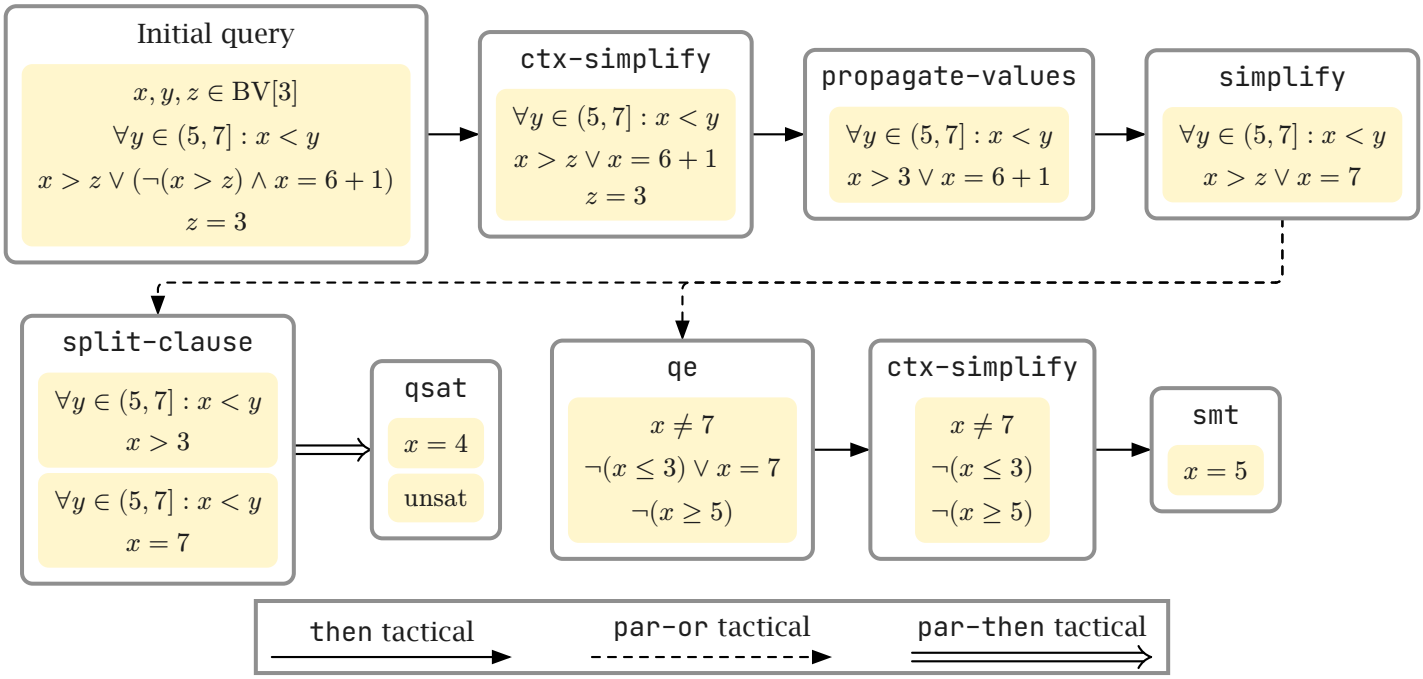


Figure 2. An example of how an SMT query would go through a pipeline of preprocessing and solver tactics, combined using various tacticals. The `par-or` tactical will try both tactics in parallel, whereas the `par-then` tactical will try to solve all goals given to it, in parallel, using the subsequent tactic.

2.2. The Core Solver

The main solver core of Z3 is exposed as the `smt` tactic; it is an implementation of a versatile and powerful SMT solver. It works with every theory that Z3 supports. It does not necessarily require any other tactics (preprocessing or otherwise) to fully solve a problem, though it is often incorporated as the last step of a more complicated architecture.

SMT queries are always a conjunction of formulas, where each formula may either be an equation in some arithmetic theory, or an arbitrarily complex propositional formula, where the propositions are arithmetic equations. Thus, boolean arithmetic is part of every SMT instance — no matter what other theories it uses — it is the top-level theory of SMT queries. For this reason, `smt` includes a SAT solver to solve for the propositional formulas[10], and a bunch of *T-solvers* which are tasked with solving for the formulas in other arithmetics.

Like most SAT solvers [2], the SAT solver in Z3 is based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [19]. That is essentially a Depth-first search (DFS) through all possible assignments of the variables, using propagation to eliminate infeasible paths. In particular, Z3 employs a Conflict-Driven Clause Learning (CDCL) [20] architecture[11], described in Algorithm 1. This is a modification of DPLL that allows the solver to learn from the conflicts that occur, which speeds up the search process. It means that the solver is able to detect conflicts earlier, so more pruning is done and the search process is faster, regardless of whether the instance is satisfiable or not. An example of how the algorithm can be applied is shown in Figure 3.

[10] The SAT instance must first be converted into Conjunctive Normal Form (CNF). In Z3, this is done using the Tseitin transformation [18].

[2] De Moura, L., and Bjørner, N. (2011) *Satisfiability Modulo Theories: Introduction and Applications*, p. 71

[19] Davis, M., Logemann, G., and Loveland, D. (1962) *A Machine Program for Theorem-Proving*

[20] Marques-Silva, J., Lynce, I., and Malik, S. (2021) *Conflict-Driven Clause Learning SAT Solvers*

[11] The solver in Z3 is more precisely called CDCL(T), with the *T* denoting that it supports different theories via the aforementioned T-solvers.

Algorithm 1: Conflict-Driven Clause Learning (CDCL)

```
1 Variable and value selection
  | /* Make a tentative decision by assigning a value to some (undecided) variable. */
  | If there are no more valid variable/value selections to make, we can conclude that the
  | constraints are unsatisfiable, and we can terminate.
2 Boolean Constraint Propagation (BCP) [21] /* unit propagation */
  | /* Reduce the search space by eliminating potential variable assignments that would
  | conflict with current assignments and constraints. */
3 If all variables have been assigned:
  | We have found a solution that satisfies the problem, and can terminate.
4 If there is a conflict:
  | /* i.e., after propagation there is a variable that has no possible assignments */
5 |  Perform conflict analysis: [20]
  | | /* Go up the execution tree to find which assignments caused the conflict. */
6 |  Learn clauses /* Add a constraint, disallowing these assignments. */
7 |  Backjumping: /* In pure DPLL, backtracking is performed instead, and no conflict analysis
  |  or clause learning is done. */
8 | | Using the learned clauses, find an appropriate decision point to go back to, skipping
  | | irrelevant decisions that would conflict with the learned clause.
9 Go to Step 1.
```

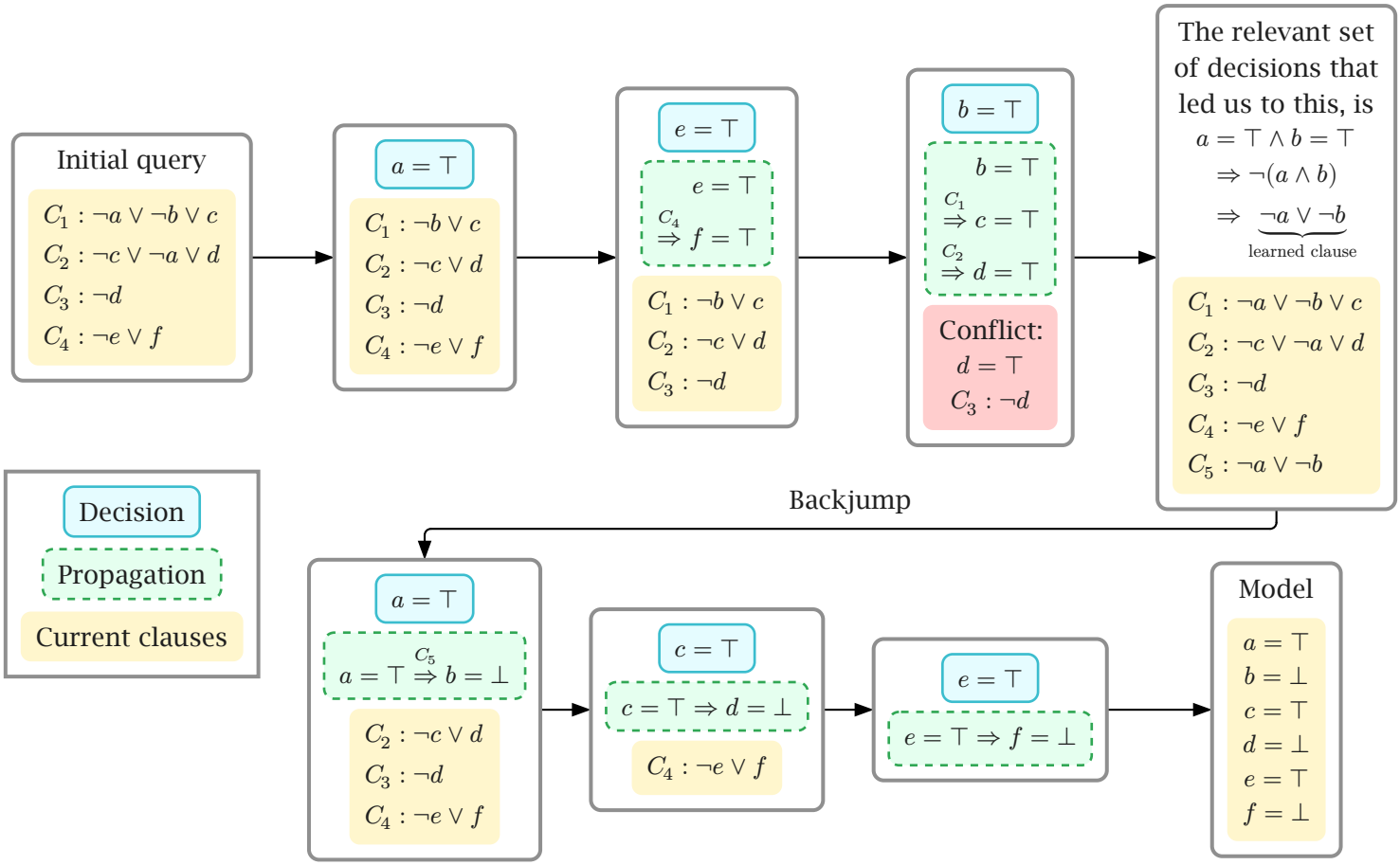


Figure 3. An example of how a CDCL algorithm can solve a given SAT query.

To deal with formulas other than the ones defined in propositional logic, Z3 includes T-solvers for each arithmetic theory. The CDCL algorithm decides on truth values for the T-formulas, and the corresponding T-solvers continuously determine whether the set of T-formulas picked by the CDCL solver can be satisfied. The default approach that Z3 uses for bit-vector formulas is called *bit-blasting*.

[21] Zhang, L., and Malik, S. (2002) *The Quest for Efficient Boolean Satisfiability Solvers*, Section 3.2.1

[20] Marques-Silva, J., Lynce, I., and Malik, S. (2021) *Conflict-Driven Clause Learning SAT Solvers*, Section 4.4

2.3. Bit-blasting

Bit-blasting transforms the problem from a bit-vector query into an equisatisfiable[12] SAT instance. The transformation treats each bit in every bit-vector as a unique propositional variable. It converts the bit-vector equations into many more formulas that work in pure propositional logic. The transformed instance is then solved[13] using the same CDCL SAT solver that Z3 uses for the boolean formulas.

Since the whole instance is fully transformed at the very beginning, this approach is more precisely categorized as *eager* bit-blasting.

[12] Either both the bit-vector query and the transformed SAT query are satisfiable, or neither of them are.

[13] If the instance is satisfiable, the propositional variables are transformed back into bit-vectors for the final output of the solver.

2.4. Lazy bit-blasting

Another bit-vector technique used by SMT solvers is *lazy* bit-blasting. The idea is that we do not perform all the transformations at the very beginning if we do not have to. Instead, we treat the problem as we would treat an instance that uses any other arithmetic theory. It is only once we need to deal with a bit-vector formula that we bit-blast it, and even then, we only bit-blast the parts of the instance that we need to continue the search. Lazy solvers can also perform certain dynamic word-level simplifications that can speed up the search.

Z3 does not natively support this approach. However, Bjørner et al. [22] have created a plug-in solution for lazy bit-blasting in Z3, called PolySAT. It is not available as a new tactic in Z3, but is rather invoked with the `smt.bv.solver=1` parameter and the `smt` tactic.

[22] Rath, J., Eisenhofer, C., Kaufmann, D., et al. (2024) *PolySAT:Word-level Bit-vector Reasoning in Z3*

2.5. Int-blasting

While bit-blasting is an effective solution, especially if a lot of bit-wise operations are involved, problems with lots of integer arithmetic may prove to be a challenge [22, 23]. This is particularly true when dealing with non-linear integer arithmetic[14], where bit-blasting can really struggle.

For this reason, a technique called *int-blasting* [24] was developed for the cvc5 SMT solver [25]. Instead of converting the bit-vectors to many boolean predicates, we convert them to integer variables, on which a solver for integer arithmetic can be used. Formulas and operations related to bit-vector integer arithmetic can directly be turned into ones that work on integer variables[15]. However, in SMT, integer variables do not have any bounds, whereas bit-vectors have a fixed length, and as such represent integers that can underflow and overflow. Thus, more constraints need to be added to take care of this, in case it happens. Moreover, operations that involve boolean arithmetic, e.g., bit-wise `and`, cannot be tackled by an integer solver natively, so they need to be simulated using integer arithmetic.

An int-blasting technique has also been implemented in Z3[16]. In particular, a lazy int-blasting approach is taken, where the bit-vectors are interpreted as unsigned integers. This corresponds most closely to the so-called lazy *sum* approach as implemented by Zohar et al. [24]. Like PolySAT,

[23] Reisenberger, C. (2014) *PBoolector: A Parallel SMT Solver for QF_BV by Combining Bit-Blasting with Look-Ahead*, Section 2.3.1

[22] Rath, J., Eisenhofer, C., Kaufmann, D., et al. (2024) *PolySAT:Word-level Bit-vector Reasoning in Z3*, p. 1

[14] Even something like the multiplication of two variables can pose a problem in SMT queries. This is an issue, given that one of the main applications of SMT solvers is software verification; of course integer arithmetic comprises a large amount of most programs.

[15] The `bv2int` and `int2bv` SMT operations are used by the solver to aid in preprocessing tactics like `simplify`.

[24] Zohar, Y., Irfan, A., Mann, M., et al. (2022) *Bit-Precise Reasoning via Int-Blasting*

[25] Barbosa, H., Barrett, C., Brain, M., et al. (2022) *Cvc5:A Versatile and Industrial-Strength SMT Solver*

[16] Int-blasting has not been documented officially, but it was added in Z3 v.4.12.5 as per the release notes.

int-blasting isn't yet available as a tactic in Z3, but is rather invoked via `smt.bv.solver=2`.

2.6. Stochastic Local Search

One last technique that can be used for bit-vector instances is Stochastic Local Search (SLS). This entails keeping a candidate solution, which does not necessarily satisfy all constraints, and mutating it, until it becomes a valid solution. This process may work well if we have a good candidate solution, but it also may not ever find an actual solution. In fact, if the instance is unsatisfiable, SLS will never be able to find that out [26].

[26] Hamadi, Y., and Sais, L., Eds. (2018) *Handbook of Parallel Constraint Reasoning*, Section 1.2.2

In Z3, SLS can be used with `smt.sls.enable=true`. SLS can work in tandem with a bit-blasting solver, since it is rarely effective completely on its own[17]. It can continuously generate candidate solutions, taking into account current assignments from the main solver to reduce its search space. If it happens upon a solution, the main solver can terminate early. Even so, it is unable to do word-level mutations, which is a major limitation especially when dealing with large bit-widths. The only mutations supported are random bit-flips.

[17] The search space can be enormous for large problems, so the chances of finding a suitable solution would be quite low.

3. Parallelism in Z3

3.1. Cube-and-Conquer

The main way that Z3 utilizes multicore processors is using the cube-and-conquer [27, 28] method. This is a divide-and-conquer approach, which involves splitting the search space into multiple subproblems, called *cubes*. To determine the best places to split the problem, a technique called *look-ahead* [29] is used.

[27] Heule, M. J. H., Kullmann, O., Wieringa, S., et al. (2012) *Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads*

[28] Heule, M. J. H., Kullmann, O., and Biere, A. (2018) *Cube-and-Conquer for Satisfiability*

[29] Heule, M. J. H., and Maaren, H. van. (2009) *Look-Ahead Based SAT Solvers*

The different parts of the problem can then be solved on different threads, independently of each other. What is more, since the solver uses the CDCL architecture, the different threads can also share learned clauses with each other, which helps to improve efficiency.

In Z3, cube-and-conquer can be activated with `parallel.enable=true`. However, the cubing strategy features many parameters that are said to significantly influence its performance [15]. We explore those parameters in Section 4.3.2.

[15] Nikolaj Bjørner, Clemens Eisenhofer, Arie Gurfinkel, et al. *Z3 Internals (Draft)*

3.2. Portfolio solving

The second main parallel technique that is often used in SAT solvers is *portfolio solving* [30]. It entails concurrently running multiple solvers or solver configurations. The justification for this may not be entirely obvious, since we are already able to parallelize with cube-and-conquer. It is, however, quite conceivable that certain solvers[18] could more quickly find a solution for a specific instance or class of instances. If this effect is strong, it may be beneficial to crudely waste computation by running multiple solvers, especially if we are unable to determine which solver to use on a particular instance ahead of time[19].

[30] Wintersteiger, C. M., Hamadi, Y., and Moura, L. de. (2009) *A Concurrent Portfolio Approach to SMT Solving*

[18] For satisfiable problems, even simply concurrently running multiple instances of the same solver with different random seeds can produce a model somewhat more quickly. In Z3, this is possible via the `sat.threads=...` parameter.

[19] Z3 does support a feature called *probes* (otherwise called *formula measures* in the literature), which allow us to change the tactics being used based on the given query. However, this feature is rather limited, mainly only allowing you to discern between different logics.

Portfolio solving in Z3 is supported using the `par-or` tactical, which can run multiple tactics on the same goal at the same time. If one of them succeeds in giving an answer (either satisfiable or unsatisfiable), the whole solver terminates with that answer. An alternative method of portfolio solving would be to simply run multiple solvers or configurations as multiple different processes.

3.3. Parallel Stochastic Local Search

Even though SLS can be ran in tandem with CDCL, one may observe that the two algorithms are independent. This means that SLS can be ran in parallel, on a completely different thread from the main algorithm [26]. As the search space is updated, the parallel SLS algorithm can continuously keep track of that and adjust accordingly.

[26] Hamadi, Y., and Sais, L., Eds. (2018) *Handbook of Parallel Constraint Reasoning*, Section 1.5

Z3 supports Parallel SLS with `smt.sls.parallel=true`, which is actually turned on by default if one is running SLS on Z3 with `smt.sls.enable=true`. In fact, with `sat.local_search_threads=...`, multiple SLS instances with different seeds can be ran at the same time in the hopes that at least one of them would find a solution.

4. Experimental Evaluation

To answer the question posed in Section 1.1.2, we compare Z3's default bit-vector algorithm — eager bit-blasting (`bit-blast`) — with the alternate methods that we described in Section 2 — lazy bit-blasting (`polysat`), int-blasting (`int-blast`), and eager bit-blasting sequentially combined with SLS (`sls`) — across different datasets.

To determine how Z3 scales with multicore CPUs (Section 1.1.3), we first perform a random search with 500 parallel configurations that use eager bit-blasting[20]. We used different numbers of threads dedicated to cube-and-conquer and parallel local search, and we tried different parameters for cube-and-conquer. We omitted portfolio solving from our experiments, though we can still make conclusions about its potential effectiveness from all the experiments that we did perform. After picking the one best parallel configuration we experimented by running it on different numbers of cores, to see how well it scales.

[20] As we will see in Section 4.3.1, eager bit-blasting with SLS generally performed the best. This is why we used it for our parallelization experiments, rather than int-blasting or lazy bit-blasting.

The metric we use to measure performance is the elapsed real time (wall time) that a configuration takes to solve a problem. This is the natural thing to do, since SMT solvers should eventually be able to solve any given query[21]. All valid solutions[22] are considered equally valuable. Aside from speed, factors like memory consumption are also recorded but are not often consequential.

[21] This holds for bit-vector arithmetic, as well as for linear number arithmetics. Beyond `QF_LIA` the problems might not be decidable, though there are solvers that employ heuristics that may prove unsatisfiability in some cases.

[22] SMT solvers that produce even a single wrong answer are usually disqualified from competitions.

4.1. Setup

We test the following configurations[23]:

bit-blast `z3 -smt2`
polysat `z3 sat.smt=true smt.bv.solver=1 -smt2`
int-blast `z3 sat.smt=true smt.bv.solver=2 -smt2`
sls `z3 smt.sls.enable=true smt.sls.parallel=false -smt2`

[23] As per [22], the polysat configuration is ran on Z3 compiled from the `poly` branch of the Z3 repository. The rest of the configurations are all ran on Z3 4.15.1.

To ensure consistency, we use a standard set of preprocessing tactics for all solver configurations (Figure 4).

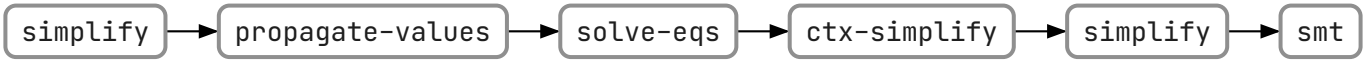


Figure 4. These tactics, combined using the `then` tactical, constitute the Z3 strategy configuration for our experiments.

The aforementioned configurations are each ran [31] on one core of an Intel Xeon E5-6248R processor (24 cores at 3.0 GHz). The parallelization tests were ran on an HPC node with two Intel Xeon E5-6448Y (32 cores at 2.1 GHz) CPUs. For the parallelization search we gave each configuration 8 cores to work with.

[31] (DHPC), D. H. P. C. C. (2024) *DelftBlue Supercomputer (Phase 2)*

To ensure the integrity of the performance measurements, we used SLURM's[24] `srun` command. This allows us to dedicate processor cores to each benchmark/solver configuration pair[25], as well as to limit memory usage and timeout instances that are taking too long. Resource usage is recorded using the built-in Linux `time` command. The

[24] Simple Linux Utility for Resource Management

[25] We also randomzie the order in which benchmark/solver pairs are ran, in order to reduce any bias in the computation.

solver is ran in a container[26] using Apptainer[27] with Linux kernel v. 4.18.0. For the purposes of reproducibility and future research, a repository with all benchmarking and data analysis code, as well as the results of the experiments, is provided on GitHub.

[26] A Docker image with the compiled versions of Z3 used in our experiments is available on Docker Hub.

[27] Apptainer (f.k.a. Singularity) - Portable, Reproducible Containers

4.2. Datasets

The benchmarking of solvers is widespread in the SMT community. There is an almost standard set of benchmarks, published every year[28]. It is a collection of user-submitted problems or families of problems for different logics, organized as part of the SMT-LIB initiative [3]. Most SMT papers use at least a subset of those benchmarks. In fact, every year an SMT solver competition called SMT-COMP [32] is held, which uses such a subset.

[28] SMT-LIB Releases on Zenodo

[3] Barrett, C., Fontaine, P., and Tinelli, C. (2016) *The Satisfiability Modulo Theories Library (SMT-LIB)*

[32] Barrett, C., Moura, L. de, and Stump, A. (2005) *SMT-COMP: Satisfiability Modulo Theories Competition*

All of this is facilitated thanks to the SMT-LIB standard [33], which serves as a common language between solvers, such that they can all use the exact same benchmarks. This has allowed for the proliferation of different datasets of benchmarks created by the SMT community. The Z3 executable supports SMT-LIB 2 as an input format. Thus, for our experiments, we use various datasets that already exist, as described in below.

[33] Barrett, C., Fontaine, P., and Tinelli, C. (2025) *The SMT-LIB Standard:Version 2.7*

For SMT-COMP, we picked the same time limits as the ones imposed by the rules of the competition. For everything else, we gave as much resources as possible to the solvers, but we capped them to a consistent amount for fairness between the solvers (summarized in Table I).

Table I. The time limits (TL) and memory limits (ML) imposed on each instance/solver pair.

Dataset	TL	ML
SMT-COMP 2024	20 min.	3 GiB
VLSAT-3	40 min.	3 GiB
Smart Contracts	1 hour	12 GiB

4.2.1. SMT-COMP 2024

First, we chose to use the benchmarks included in the 2024 edition of SMT-COMP[29]. In particular, we used the Single-Query `QF_Bitvec` track, which contains 10703 instances. They are divided into families of benchmarks based on their origin. This dataset is a representative sample of the full SMT-LIB `QF_BV` dataset, created by proportionally taking more instances from smaller families of benchmarks. This is also the dataset that we used in our parallelization experiments.

[29] *19th International Satisfiability Modulo Theories Competition (SMT-COMP 2024)*

4.2.2. VLSAT-3

Second, we use the Very Large Boolean SATisfiability (VLSAT) benchmark set [34] to specifically test large instances, related to the verification of "communication protocols, distributed systems and hardware circuits." VLSAT-3 *Fam-*

[34] Bouvier, P. (2021) *The VLSAT-3 Benchmark Suite*

ily a contains satisfiable `QF_BV` problems; *Family g* contains unsatisfiable ones. Each family contains 100 benchmarks.

4.2.3. Smart Contract Verification

Finally, we test a more domain-specific application of SMT solvers. Smart contracts are programs used to automatically execute cryptocurrency transactions upon certain conditions, without the need for a trusted party. Thus, it is imperative that smart contracts run as expected by all parties in the transaction. To aid in this, Albert et al. [35] developed a smart contract verification technique, which uses an SMT solver as a back-end.

[35] Albert, E., Grossman, S., Rinetzky, N., et al. (2020) *Taming Callbacks for Smart Contract Modularity*

Zohar et al. [24] showed that the int-blasting technique they developed for the cvc5 solver works better than bit-blasting, on a dataset consisting of SMT queries generated by this smart contract verification tool. The likely reason being the use of integer arithmetic with bit-widths higher than 64. We decided to try to replicate these results, with the int-blasting mode of Z3.

[24] Zohar, Y., Irfan, A., Mann, M., et al. (2022) *Bit-Precise Reasoning via Int-Blasting*

4.3. Results

To determine whether the results from our benchmarks are statistically significant, we follow the recommendations of J. Demšar [36]. First, we run a Friedman test with $\alpha(\text{p-value}) = 0.05$, which tells us whether there is any actual difference between the solver configurations. If the test is successful, we run a post-hoc critical difference analysis using the Nemenyi method, showcased via diagrams[30].

[36] Demšar, J. (2006) *Statistical Comparisons of Classifiers over Multiple Data Sets*

[30] For ease of the reader, a line is drawn through any configurations that do not have the required critical distance for statistical significance. These lines are not transitive.

The critical difference analysis is performed as follows. For every instance, we rank the solver times of each configuration (timeouts are ranked, equally, as last) with rank 1 being the best. We take the average of all ranks for each solver and plot them. The critical difference distance (CD) is plotted as well; it is calculated based on the number of instances and the desired p-value.

This approach has several advantages. For one, it allows us to simultaneously compare multiple configurations. Second, it does not assume that the data is normally distributed. Third, we are able to include instances where one or more solvers time out, whereas with other approaches we would have to completely remove such instances from the dataset.

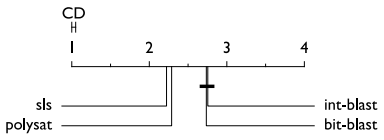


Figure 5. Critical difference between the four solver configurations on the SMT-COMP dataset.

4.3.1. Comparison of Bit-vector Techniques

In the SMT-COMP benchmark set, the very best solver is eager bit-blasting with SLS (Figure 5). A likely reason for this is the large amount of relatively easy instances in the dataset. Instances where the search space is smaller, or which have more possible solutions, are better suited to local search. Second-best is lazy bit-blasting, while its eager variant and int-blasting are both last. This does not corroborate the claim of Rath et al. [22]: "our results… indicate that there is no clear winner among existing solvers." There are two likely reasons for this: they do not perform any statistical analysis of the results, and they only use a timeout of 60 s.

As can be seen in Figure 7, the results seem to be consistent across most families in the dataset. In none of the 15 families that passed the Friedman test were pure eager bit-blasting, or int-blasting definitively the best. This holds even in the `pspace` family (Figure 6a) which features bit-widths upwards of 30 000. Lazy bit-blasting (`polysat`) is usually second, except in the `bruttomesso` family (Figure 6b). There we see that the two eager blasting methods significantly outperform both lazy approaches. That family contains synthetic benchmarks, almost entirely composed of concatenation and bit-extraction operations. The `spear` family is the only place where `polysat` is definitively the best solver. That family is composed of real-world queries generated by the Calysto static checker[31], analyzing several prominent open-source C programs for `null` pointer dereferencing. The success of `polysat` could be attributed to the especially heavy use of the if-then-else (`ite`) construct.

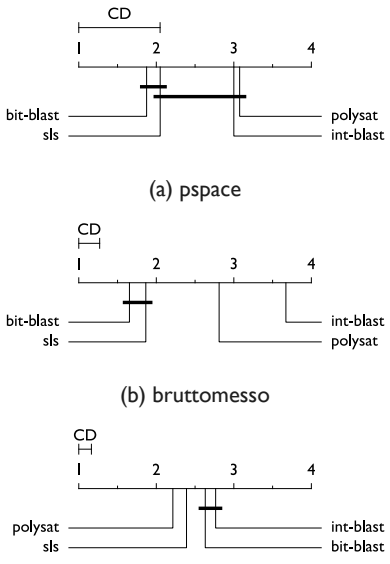


Figure 6. Crittical difference plot for three selected SMT-COMP families.

[22] Rath, J., Eisenhofer, C., Kaufmann, D., et al. (2024) *PolySAT:Word-level Bit-vector Reasoning in Z3*

[31] Calysto Software Verification Benchmarks

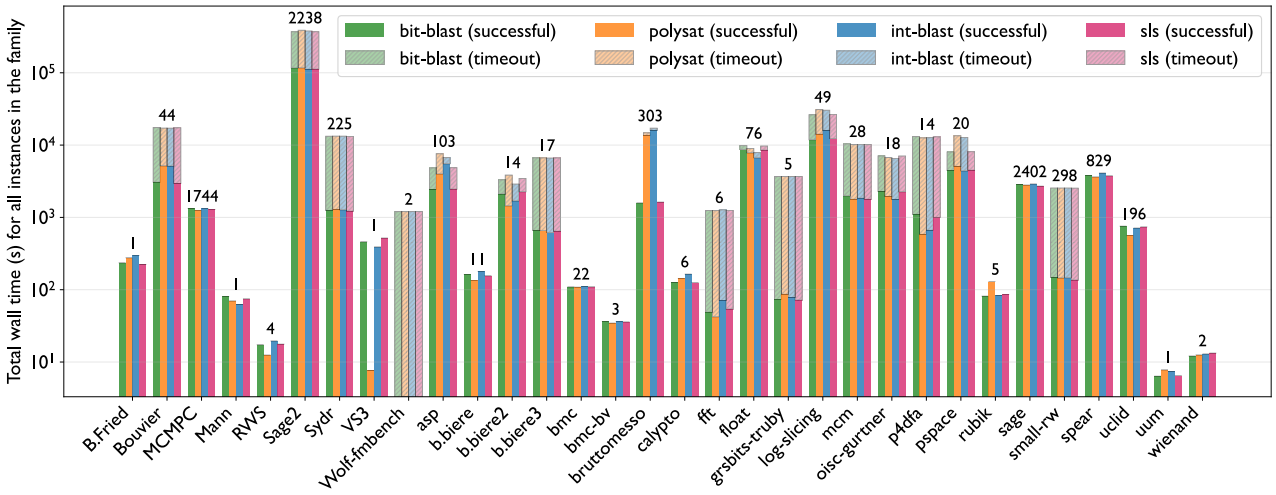
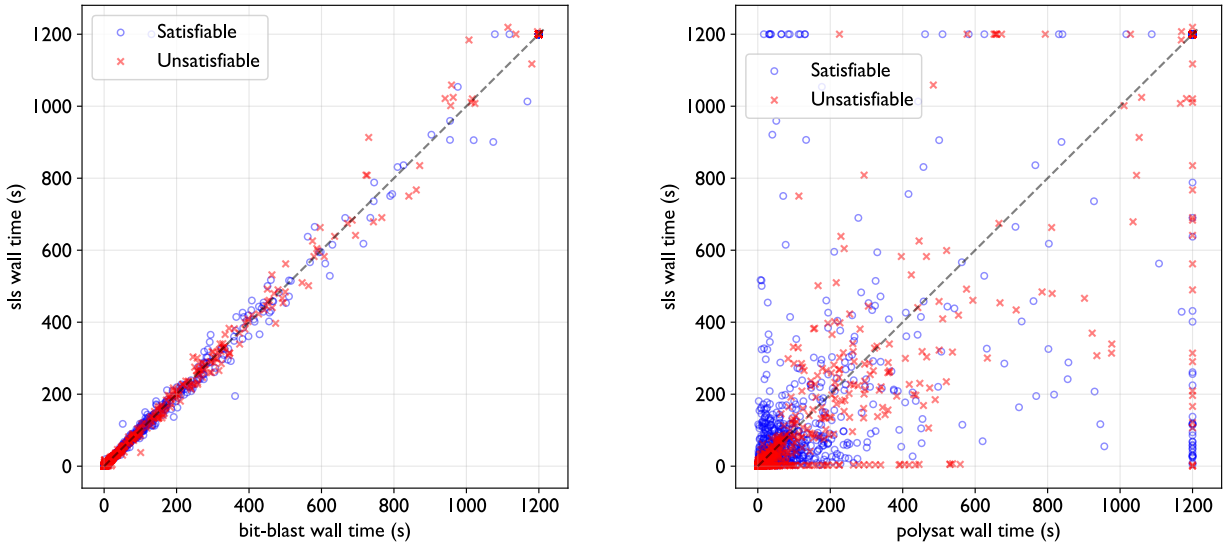


Figure 7. The performance of the four solver configurations across different families in the SMT-COMP dataset.

To finish our analysis of SMT-COMP, we compare individual pairs of solvers. Figure 8a shows that `sls` and pure `bit-blast` perform about equally on virtually all instances, even though `sls` is, on average, better. In contrast, when comparing the two best configurations for SMT-COMP (Figure 8b), there are many instances where `polysat` is faster than `sls`, and vice versa. This implies that running `polysat` and `sls` in a portfolio manner can be beneficial.



(a) `sls` vs `bitblast` (b) `sls` vs `polysat`

Figure 8. Scatterplots comparing pairs of configurations on the SMT-COMP dataset.

As for the VLSAT-3 dataset, we can see in Figure 11a that int-blasting and lazy bit-blasting always outperform the other methods in VLSAT-3 Family a. Int-blasting, in particular, solves 22 more instances than the eager bit-blast technique within the time limit. However, even int-blasting is not able to solve all problems, leaving 25 unsolved. Nevertheless, as Figure 9 confirms, lazy bit-blasting and int-blasting are both statistically better than eager bit blasting, regardless of whether SLS is enabled.

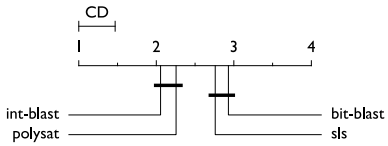


Figure 9. Critical difference between the solvers on VLSAT-3 Family a.

The same results do not appear in Family g (Figure 10), where we see that, overall, eager bit-blasting with SLS is better than both lazy bit-blasting and int-blasting. However, observing Figure 11b, shows that the effect seems to be reversed in the hardest 27 instances of the dataset, though even then `polysat` cannot solve 14 of them.

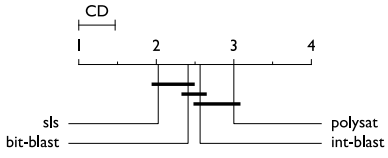
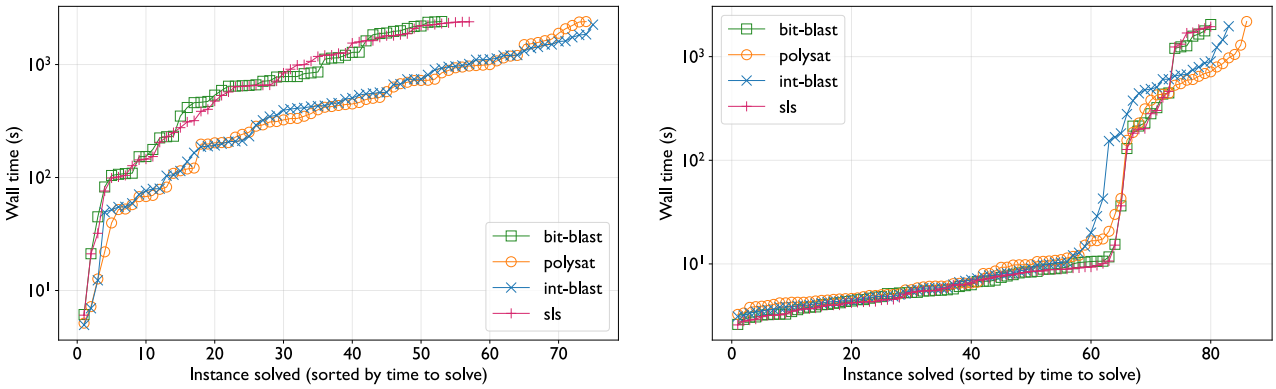


Figure 10. Critical difference between the solvers on VLSAT-3 Family g.



(a) VLSAT-3 Family a (unsatisfiable)

(b) VLSAT-3 Family g (satisfiable)

Figure 11. Cactus plots comparing the performance of the different solvers on the VLSAT-3 dataset.

Our experiments confirm the claim of Zohar et al. [24] that int-blasting inherently is "more competitive on unsatisfiable benchmarks than satisfiable ones." Int-blasting can, in a sense, function as a reduction of the bit-vector problem. It may be, even without the extra bounds constraints, that an instance is unsatisfiable. In those cases, the superior performance of Z3's integer arithmetic T-solver, when it comes to comparison operations, can prove unsatisfiability more quickly.

[24] Zohar, Y., Irfan, A., Mann, M., et al. (2022) *Bit-Precise Reasoning via Int-Blasting*

It is hard to determine ahead of time which problems are going to be satisfiable, or which ones will be easy or hard for solvers. We performed some post-hoc analysis on the SMT-COMP dataset, trying to categorize instances based on the operations or variables they contain. We tested several potential metrics: total bit-width of all variables, average bit-width, number of variables, number of operations, and number of integer arithmetic operations ($+, -, *, /$, etc.) vs. number of bit and boolean operations ($\wedge, \&, \vee$, etc.). None of those categorizations proved to be a good predictor for whether one of the solvers would perform better than the others.

Finally, on the smart contract verification tasks, we did not unfortunately get conclusive results, so we are not able to confirm the results that Zohar et al. got, where int-blasting vastly outperformed the bit-blasting solvers on this dataset. The solvers only managed to solve 7 out of the 39 instances in the `QF_BV` portion of the dataset, due to hitting memory limits. However, from Figure 12, we do see that Z3's `int-blast` solver could potentially also offer superior performance on benchmarks like these.

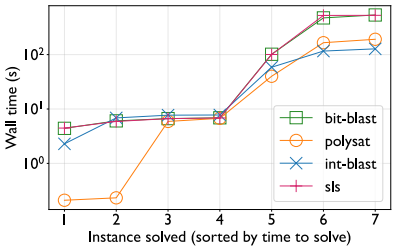


Figure 12. Cactus plot comparing the different configurations on the Smart Contract Verification dataset.

4.3.2. Parallel Scaling of Z3

After running the random search of 500 configurations on 30 instances of the SMT-COMP dataset, the top 10 best ranked solvers all happened to only use cube-and-conquer for parallelism. It appears this is a more effective strategy than parallel local search. The best solver configurations are shown in Table II. We picked the very best one to test how it scales with more cores.

Figure 13 shows that on a processor with up to 4 cores, the parallelizataion potential of cube-and-conquer does not justify the overhead that comes with the algorithm. Still, as is seen in Figure 14, on the hardest ~300 instances, the 2 and 4 core configurations do outperform the single-threaded approach. With more than 4 cores, the parallelization of cube-and-conquer really begins to shine. The peak performance is achieved with 32 cores. It seems that with 64 cores, the individual cubes become to small to be efficiently solved by the processor cores.

Table II. The top parallel configurations from our random search. The parameters `batch_size` (BS), `delay` (D), `restart.max` (R), and `backtrack_frequency` (BF) were set as follows.

Avg. rank	BS	D	R	BF
111.16	50	0	10	1
116.09	1000	5	10	100
124.62	1000	20	50	50
124.71	200	10	5	1000
128.09	500	20	25	1000

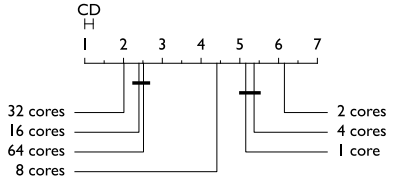


Figure 13. Critical difference analysis of Z3 configured to run with different numbers of cores.

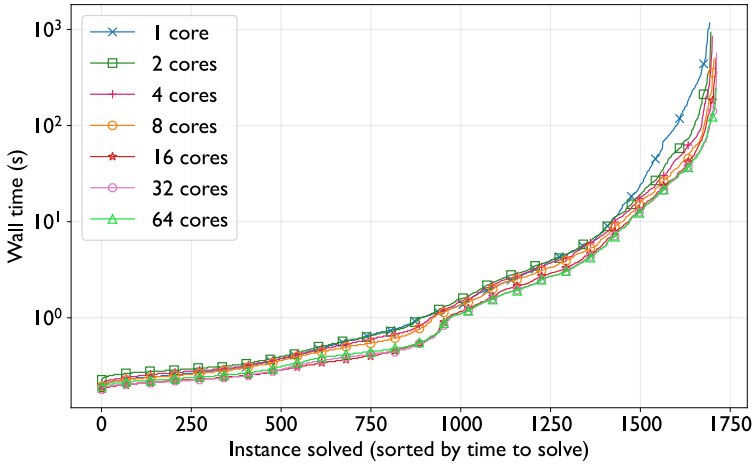


Figure 14. Cactus plot comparing how Z3 is able to utilize different numbers of CPU cores.

5. Responsible Research

This section is a discussion on the validity and importance of the research performed during the course of the project. It does not have a material impact on the results of the research.

5.1. Reproducibility and Integrity

We offer all necessary information to reproduce our experiments. The datasets that we use are all freely available. Furthermore, we provide the code we used for benchmarking and data analysis. We also provide the actual results from the benchmarks, so they can be analyzed independently. Nevertheless, our research would likely be rated only with the “Artifacts Evaluated — Functional”

badge from ACM. We have provided sufficient documentation; the artifacts are a complete and consistent representation of the research in this paper, and they are fully exercisable with the proper software setup. However, no great care was taken to completely document and organize the provided repository.

The claims made in this paper are all supported with citations and links, and the experimental results are evaluated using robust statistical approaches. This should ensure the integrity of our conclusions.

5.2. Limitations and Threats to Validity

A major limitation of this project is the computational resources that were afforded to it. For instance, a lot of papers in this field are able to include experiments that run the whole SMT-LIB benchmark suite, rather than just a subset of it. More instances, higher timeouts and memory limits, and even more runs of the same experiments can all provide more statistically significant and conclusive results. Using TU Delft's High-Performance Computing (HPC) cluster allowed us to have some such results.

As is seen in the case of the smart contract verification, this limitation directly impacted the quality of the results. Another example of this, is that we were only able to run our random search on 30 instances of SMT-COMP. With 500 configurations, we were not able to gauge the statistical significance of our random search. Therefore, we cannot say for certain that we were using the best-scaling parallel configuration in our experiments.

Aside from general lack of resources, there were some other some specific limitations on our experiments. First, we were not able to control exactly which CPU cores are allocated to our parallel benchmarking instances. This does not follow the general advice of assigning cores that are physically close together to the same benchmarked process. This is a feature of the `benchexec` tool [37], used prolifically in the SMT community, but we were not able to successfully make it work with the SLURM configuration of Delft Blue. Second, when doing our statistical analysis using the Friedman tests and critical difference diagrams, we only consider the ranking of each solver for each test instance, rather than the absolute differences in speed. If this was something particularly important to the user, then they ought to look at our quantile plots. Third, it is plausible that some preprocessing tactics work better with some solver techniques than others. For example, lazy bit-blasting and int-blasting may benefit less from aggressive preprocessing than eager bit-blasting. Because of amount of preprocessing tactics, each with dozens of parameters, we opted to not explore that in our research.

Lastly, the author of this paper acknowledges the use of LLMs (OpenAI's o4-mini, Anthropic's Claude Sonnet 4 and Sonnet 3.7 Thinking) for the benchmarking, data analysis and plotting code in the repository. It is not feasible to provide the prompts used since the models were working with and referring to a changing codebase, and the vast majority of prompts were replies to the models' outputs. This should not pose a major threat to the integrity of the research because all the code was reviewed, as well as frequently manually modified to achieve the desired functionality. No models were used in the writing of the paper.

5.3. Ethical Considerations

The main application of SMT solvers is formal verification, specifically of software, and bit-vectors play a crucial role in that. Ensuring the correctness of software has increasingly become a concern, even receiving the attention of The White House [38]. Formal verification, along with automated software testing, have established themselves as the two main tools that developers can use to this end. Z3 is a very established backend for many software verifiers. Thus, it is imperative that it works as efficiently as possible, so that software verifiers become more capable and widely adopted by developers. This underscores the importance of the research in this field.

Typical ethical considerations, such as the handling of sensitive end-user data or the prevention of bias in automated decision-making, do not apply to our research. Most of the benchmarks used in our experiments are synthetically generated, but some do originate from real-world applications of SMT solvers. We trust that the authors of these benchmarks and the organizers of the SMT-LIB benchmark collection have employed due diligence in the creation of the datasets.

6. Conclusion and Future Work

Z3 is a very capable SMT solver that handles bit-vector problems well. It features four alternative techniques to this end: eager and lazy bit-blasting, int-blasting, and local search, though not all of them have been fully integrated into the solver. The latest techniques, int-blasting and lazy bit-blasting, are not fully supported as tactics in Z3′s powerful and unique way of strategy configuration, though that is not necessarily a major issue.

Z3′s robust conflit-driven clause learning SAT solver allows even the crude eager bit-blasting approach to be efficient. When sequentially paired with stochastic local search, eager bit-blasting seems to the best algorithm in most cases, but particularly on easier instances. Nevertheless, we have shown that there are problem sets where other techniques can shine. For one, there are many instances where lazy bit-blasting solves vastly outperforms its eager variant, even if it is not better on average. What is more, on sets of queries, generated by certain tools, lazy bit-blasting can offer a significant advantage. Finally, the int-blasting approach, while outperformed in most cases, can turn out to be a valuable tool, in harder problems or unsatisfiable ones.

Our general recommendation for developers using Z3 as a backend is to test all four bit-vector solving techniques on a set of queries generated by their tools. Then, they can analyze them using the methods we described in this paper to find out the best technique for their specific use case. This may lead to a significant performance increase.

Z3 employs three main concurrency techniques, which can all work in tandem: cube-and-conquer, portfolio solving, and parallel local search. The results from our experiments on the parallel scalability of Z3 show that for users with less than 4 processor cores the solver is best ran on just a single one, unless the problems being solved are very hard. If more than 4 cores are afforded, the cube-and-conquer approach scales well, but at 64 cores it starts to suffer. At that point, if the solve time of individual queries is a top priority, one ought to run multiple solvers in parallel via a portfolio solving approach. Lazy bit-blasting and eager bit-blasting with local search, are good candidates for this.

In the future, int-blasting and lazy bit-blasting can be fully documented and integrated into Z3, including developing them as standalone tactics. This would allow for further research into the usefulness of these techniques. They can also be integrated as part of `Z3alpha`, which could automate the process of categorizing the specific instances where each technique shines. Alternatively, machine learning approaches could be used to possibly the same effect. Parallelization schemes and parameters can also be included in automated configuration searches. Moreover, work can be done to better classify SMT queries, based on properties like bit-width, or the types of operations in them, which could give new insights into when exactly certain solvers work best. Finally, the practicality of portfolio solving with the four different bit-vector algorithms can be analyzed using the data from our experiments.

References

[1] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer, 2008, pp. 337–340. doi: 10.1007/978-3-540-78800-3_24.

[2] L. De Moura and N. Bjørner, "Satisfiability modulo Theories: Introduction and Applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011, doi: 10.1145/1995376.1995394.

[3] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability modulo Theories Library (SMT-LIB)." [Online]. Available: https://smt-lib.org/

[4] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-Based Bounded Model Checking for Embedded ANSI-C Software." Accessed: Jun. 19, 2025. [Online]. Available: https://arxiv.org/abs/0907.2072v2

[5] P. Godefroid, M. Y. Levin, D. A. Molnar, and others, "Automated Whitebox Fuzz Testing.," in *NDSS*, 2008, pp. 151–166.

[6] D. Kroening and O. Strichman, "Applications in Software Engineering and Computational Biology," *Decision Procedures: An Algorithmic Point of View*. Springer, Berlin, Heidelberg, pp. 281–307, 2016. doi: 10.1007/978-3-662-50497-0_12.

[7] N. Bjørner, M. Levatich, N. P. Lopes, A. Rybalchenko, and C. Vuppalapati, "Supercharging Plant Configurations Using Z3," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, P. J. Stuckey, Ed., Cham: Springer International Publishing, 2021, pp. 1–25. doi: 10.1007/978-3-030-78230-6_1.

[8] A. Nadel and V. Ryvchin, "Bit-Vector Optimization: 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2016 and Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016," *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016 and Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Proceedings*, pp. 851–867, 2016, doi: 10.1007/978-3-662-49674-9_53.

[9] A. C. Wright, "Modular SMT-Based Verification of Rule-Based Hardware Designs," 2021. Accessed: Jun. 19, 2025. [Online]. Available: https://dspace.mit.edu/handle/1721.1/139491

[10] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks." Accessed: Jun. 19, 2025. [Online]. Available: http://arxiv.org/abs/1702.01135

[11] B. Schauer, "Multicore Processors – A Necessity," *ProQuest discovery guides*, vol. 59, 2008.

[12] T. Guertin and A. Hurson, "The Multicore Architecture," *Advances in Computers*, vol. 130. Elsevier, pp. 139–162, Jan. 01, 2023. doi: 10.1016/bs.adcom.2022.09.003.

[13] R. S. Ramesh, "Scalable Systems and Software Architectures for High-Performance Computing on Cloud Platforms." Accessed: Jun. 22, 2025. [Online]. Available: http://arxiv.org/abs/2408.10281

[14] S. Stoller *et al.*, "Future Directions for Parallel and Distributed Computing," in *2019, Report of an NSF Workshop to Influence the Successor to the Scalable Parallelism in the Extreme (SPX) Program*, 2019.

[15] Nikolaj Bjørner *et al.*, "Z3 Internals (Draft)." [Online]. Available: https://z3prover.github.io/papers/z3internals.html

[16] Z. Lu, S. Siemer, P. Jha, J. Day, F. Manea, and V. Ganesh, "Layered and Staged Monte Carlo Tree Search for SMT Strategy Synthesis." Accessed: Jun. 05, 2025. [Online]. Available: http://arxiv.org/abs/2401.17159

[17] L. de Moura and G. O. Passmore, "The Strategy Challenge in SMT Solving," *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*. Springer, Berlin, Heidelberg, pp. 15–44, 2013. doi: 10.1007/978-3-642-36675-8_2.

[18] G. S. Tseitin, "On the Complexity of Derivation in Propositional Calculus," *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Springer, Berlin, Heidelberg, pp. 466–483, 1983. doi: 10.1007/978-3-642-81955-1_28.

[19] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-Proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962, doi: 10.1145/368273.368557.

[20] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-Driven Clause Learning SAT Solvers," *Handbook of Satisfiability*. IOS Press, pp. 133–182, 2021. doi: 10.3233/FAIA200987.

[21] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," in *Computer Aided Verification*, E. Brinksma and K. G. Larsen, Eds., Berlin, Heidelberg: Springer, 2002, pp. 17–36. doi: 10.1007/3-540-45657-0_2.

[22] J. Rath, C. Eisenhofer, D. Kaufmann, N. Bjørner, and L. Kovács, "PolySAT: Word-level Bit-vector Reasoning in Z3." Accessed: May 20, 2025. [Online]. Available: http://arxiv.org/abs/2406.04696

[23] C. Reisenberger, "PBoolector: A Parallel SMT Solver for QF_BV by Combining Bit-Blasting with Look-Ahead," Linz, 2014.

[24] Y. Zohar *et al.*, "Bit-Precise Reasoning via Int-Blasting," in *Verification, Model Checking, and Abstract Interpretation*, B. Finkbeiner and T. Wies, Eds., Cham: Springer International Publishing, 2022, pp. 496–518. doi: 10.1007/978-3-030-94583-1_24.

[25] H. Barbosa *et al.*, "Cvc5: A Versatile and Industrial-Strength SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds., Cham: Springer International Publishing, 2022, pp. 415–442. doi: 10.1007/978-3-030-99524-9_24.

[26] Y. Hamadi and L. Sais, Eds., *Handbook of Parallel Constraint Reasoning*. Cham: Springer International Publishing, 2018. doi: 10.1007/978-3-319-63516-3.

[27] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads," in *Hardware and Software: Verification and Testing*, K. Eder, J. Lourenço, and O. Shehory, Eds., Berlin, Heidelberg: Springer, 2012, pp. 50–65. doi: 10.1007/978-3-642-34188-5_8.

[28] M. J. H. Heule, O. Kullmann, and A. Biere, "Cube-and-Conquer for Satisfiability," *Handbook of Parallel Constraint Reasoning*. Springer International Publishing, Cham, pp. 31–59, 2018. doi: 10.1007/978-3-319-63516-3_2.

[29] M. J. H. Heule and H. van Maaren, "Look-Ahead Based SAT Solvers," *Handbook of Satisfiability*. IOS Press, pp. 155–184, 2009. doi: 10.3233/978-1-58603-929-5-155.

[30] C. M. Wintersteiger, Y. Hamadi, and L. de Moura, "A Concurrent Portfolio Approach to SMT Solving," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds., Berlin, Heidelberg: Springer, 2009, pp. 715–720. doi: 10.1007/978-3-642-02658-4_60.

[31] D. H. P. C. C. (DHPC), "DelftBlue Supercomputer (Phase 2)." [Online]. Available: https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2

[32] C. Barrett, L. de Moura, and A. Stump, "SMT-COMP: Satisfiability Modulo Theories Competition," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds., Berlin, Heidelberg: Springer, 2005, pp. 20–23. doi: 10.1007/11513988_4.

[33] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.7," 2025.

[34] P. Bouvier, "The VLSAT-3 Benchmark Suite." Accessed: Jun. 05, 2025. [Online]. Available: http://arxiv.org/abs/2112.03675

[35] E. Albert, S. Grossman, N. Rinetzky, C. Rodríguez-Núñez, A. Rubio, and M. Sagiv, "Taming Callbacks for Smart Contract Modularity," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–30, Nov. 2020, doi: 10.1145/3428277.

[36] J. Demšar, "Statistical Comparisons of Classifiers over Multiple Data Sets," *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, Dec. 2006.

[37] D. Beyer, S. Löwe, and P. Wendler, "Reliable Benchmarking: Requirements and Solutions," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 1, pp. 1–29, Feb. 2019, doi: 10.1007/s10009-017-0469-y.

[38] T. W. House, "Back to the Building Blocks: A Path toward Secure and Measurable Software." 2024.