

# Design and Evaluation of a Formula Cache for SMT-based Bounded Model Checking Tools

by

Jean Anré Breytenbach



UNIVERSITEIT  
STELLENBOSCH  
UNIVERSITY

**100**  
1918 · 2018

*Thesis presented in partial fulfilment of the requirements for the degree of  
Master of Science in Computer Science in the Faculty of Science at  
Stellenbosch University*

Supervisor: Prof. Bernd Fischer

March 2018

The financial assistance of the Council of Scientific and Industrial Research (CSIR) through the Centre for Artificial Intelligence Research (CAIR) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the CSIR.

# Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: ..... March 2018

Copyright © 2018 Stellenbosch University  
All rights reserved.

# Abstract

## **Design and Evaluation of a Formula Cache for SMT-based Bounded Model Checking Tools**

Jean A. Breytenbach

*Division of Computer Science,  
Department of Mathematical Sciences,  
University of Stellenbosch,  
Private Bag X1, 7602 Matieland, South Africa.*

Thesis: MSc Computer Science

March 2018

Program verification is a computationally expensive and time-consuming process. *Bounded model checking* is a branch of program verification that produces FOL formulas to be checked for satisfiability by an SMT solver. These formulas encode state transitions to states where property violations will occur and the *SMT solver* attempts to find a list of variable assignments that would create a path to one of these states. Bounded model checking tools create these formulas by iteratively increasing an unwind bound  $k$  that dictates the number of transitions that can be present in a path, for each unwind bound  $k$  all possible paths of length  $k$  are generated. Any state containing a property violation that is generated during the unwind bound  $k - 1$  should also be present during the unwind bound  $k$  with perhaps a longer path to reach it.

This creates many of the same paths being checked during each subsequent iteration causing the SMT solver to potentially perform duplicate work. This thesis seeks to build and evaluate a formula cache in which to store parts of the formula for which the satisfiability is already known. During subsequent iterations the formula can be sliced by removing the parts that are found in the cache, providing smaller formulas for the SMT solver which should take less time to solve. Similar formula caches have already proven successful in the field of *symbolic execution*.

Multiple techniques are described to modify the formulas to increase the likelihood of finding a match in the cache and these techniques are combined in a multitude of ways to generate a collection of caching strategies. These strategies are then evaluated against multiple data sets to find the best performing strategies and to identify the types of problems that benefit the most from caching. The results are then compared to the results of caching formulas during symbolic execution to gain insight as to how these different approaches effectively implement caching.

# Uittreksel

## Ontwerp en Evaluering van 'n Formule Kasstelsel vir SMT-gebaseerde Begrensde Modeltoetsing Gerdeedskap

Jean A. Breytenbach

Afdeling Rekenaarwetenskap,  
Department van Wiskundige Wetenskap,  
Universiteit van Stellenbosch,  
Privaatsak X1, 7602 Matieland, Suid Afrika.

Tesis: MSc Rekernaarwetenskap

Maart 2018

Program verifikasie is 'n duur en tydrowende berekeningsproses. *Begrensde modeltoetsing* is 'n tak van program verifikasie wat FOL-formules genereer en toets vir voldoening deur gebruik te maak van 'n *SMT oplosser*. Hierdie formules koördineer die paaie deur programme waar eienskappe hul waardes verkry buite die bepaalde spesifikasies. Begrensde modeltoetsing gereedskap bou hierdie formules deur iteratief 'n bogrenswaardewaarde  $k$ , wat die aantal stappe wat in 'n pad teenwoordig kan wees, te verhoog. Vir elke bogrenswaarde  $k$  word alle moontlike paaie van lengte  $k$  gegeneer.

Indien 'n formule op 'n sekere pad nie voldoen aan die spesifikasie vir  $k - 1$  stappe nie, sal daardie gedeelte van die formule nogsteeds ongeldig wees vir  $k$  stappe. Dus word baie van dieselfde paaie tydens elke opeenvolgende iterasie gegenereer. Gevolglik word baie van die werk gedupliqueer vir die SMT oplosser tydens opeenvolgende iterasies. Hierdie tesis poog om 'n formulekas te bou en te evalueer, wat dele van die formule op kyk waarvoor die voldoening reeds bekend is. Tydens opeenvolgende iterasies kan die formule gesny word deur van die gedeeltes te verwys wat in die formulekas gevind kan word. Soortgelyke formulekasstelsels is reeds suksesvol bewys in die gebied van simboliese uitvoering.

'n Aantal tegnieke word beskryf om die formules te verander gedurende die opkyk proses. Daar word verwag dat die transformasies die waarskynlikheid om 'n ekwivalente formule in die formulekas te vind, sal verhoog. Die tegnieke word gekombineer in veelvuldige strategieë om dan te evalueer watter kombinasies die beste resultate lewer vir begrensde modeltoetsing.

# Acknowledgements

Thank you to my supervisor, Bernd Fischer, for guiding me through my projects during my Honours and Masters at Stellenbosch University, giving me the necessary support and feedback for the completion of my work.

Thank you to Willem Visser and Jaco Geldenhuys for their consultation and insight with regards to the operation of Green.

Some computations were performed using the University of Stellenbosch's HPC1 (Rhasatsha): <http://www.sun.ac.za/hpc>

Thank you to Mikhail Ramalho from the University of Southampton, for giving advice with regards to the integration of the cache with ESBMC as well as providing experimental results.

The author acknowledges the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work.

Thank you to my friends that have toiled by my side into the early morning hours for the past 6 years.

# Dedication

*For my parents, Manja and Marcel Breytenbach.*

*Some accidents turn out to be successful ones...*

# Contents

<b>Declaration</b>	ii
<b>Abstract</b>	iii
<b>Uittreksel</b>	iv
<b>Acknowledgements</b>	v
<b>Dedication</b>	vi
<b>Contents</b>	vii
<b>List of Figures</b>	x
<b>List of Tables</b>	xii
<b>1 Introduction</b>	1
<b>2 Background</b>	3
2.1 Model Checking . . . . .	3
2.2 Symbolic Model Checking . . . . .	4
2.3 Bounded Model Checking . . . . .	5
2.4 Existing Formula Caches . . . . .	8
2.5 SMT Solvers . . . . .	10
2.5.1 boolector . . . . .	10
2.5.2 Z3 . . . . .	10
2.5.3 Solver Unpredictability . . . . .	10
<b>3 Cache Design</b>	15
3.1 Goals . . . . .	15
3.1.1 Decreasing End-To-End Time Of ESBMC . . . . .	15
3.1.2 Decreasing Solver Time And Increasing Cache Hits . . . . .	15
3.1.3 Across-Run Reuse . . . . .	16
3.2 Understanding ESBMC Formulas . . . . .	16
3.3 Handling Asserts Individually . . . . .	19
3.4 Formula Representation . . . . .	20
3.5 Intercepting formulas between ESBMC and SMT Solvers . . . . .	23
3.6 Storing Formulas For Exact Matches . . . . .	24
3.7 Storing Formulas For Subset Matches . . . . .	25

3.7.1	Proving a Subset Match . . . . .	25
3.7.2	Decreasing The Size Of $G$ . . . . .	26
3.7.3	Finding A Subset Match . . . . .	27
3.7.4	Filters . . . . .	28
3.7.5	ESBMC Program Options . . . . .	33
3.8	Unsatisfiable Cores . . . . .	33
3.9	Store formulas before results are known . . . . .	36
3.10	Storing Incorrect Results . . . . .	38
<b>4</b>	<b>Formula Modifications For Increased Reuse Potential</b>	<b>41</b>
4.1	Expressions Relevant To Assert Only . . . . .	41
4.2	Renaming Variables within Formulas . . . . .	42
4.3	Variable Dependent Formulas . . . . .	43
4.4	Canonization . . . . .	44
4.5	Variable Propagation . . . . .	45
4.5.1	Single assignment . . . . .	45
4.5.2	Multiple assignments . . . . .	45
4.6	Variable Simplification . . . . .	46
4.7	Formula Simplification . . . . .	47
4.7.1	Simplify Replaced . . . . .	47
4.7.2	Simplify . . . . .	48
4.7.3	Simplify During . . . . .	48
4.8	Splitting Assumptions From Asserts . . . . .	48
<b>5</b>	<b>Evaluating Cache Performance</b>	<b>51</b>
5.1	Experimental Setups and Overview . . . . .	51
5.2	Interpreting data from graphs . . . . .	52
5.3	Strategies . . . . .	55
5.4	Data Set 1 . . . . .	56
5.4.1	Setup . . . . .	56
5.4.2	Overview of results . . . . .	57
5.4.3	No modifications and Variable Renaming . . . . .	64
5.4.4	Slicing Assignments . . . . .	67
5.4.5	Formula Modifications . . . . .	69
5.4.6	Storing formulas before their satisfiability is known . . . . .	71
5.4.7	Subset Filters . . . . .	72
5.4.8	Formula modifications . . . . .	74
5.4.9	Unsat Core Lookup . . . . .	76
5.4.10	Hybrid Strategies . . . . .	78
5.4.11	Identifying significant overheads for Strategy 20 and 22 . . . . .	80
5.4.12	Identifying categories where caching works better . . . . .	82
5.5	Data Set 2 . . . . .	86
5.5.1	Setup . . . . .	86
5.5.2	Results . . . . .	87
5.5.3	Insights . . . . .	92
5.6	Data Set 3 . . . . .	93
5.6.1	Setup . . . . .	93
5.6.2	Results . . . . .	93
5.6.3	Insight . . . . .	99
5.7	Data Set 4 . . . . .	100

5.7.1	Setup . . . . .	100
5.7.2	Results . . . . .	100
<b>6</b>	<b>Conclusions</b>	<b>107</b>
6.1	Symbolic Execution vs Bounded Model Checking . . . . .	107
6.2	Effectiveness of caching formulas for Bounded Model Checking . . . . .	110
6.3	Influence of caching formulas produced by BMC on SMT solvers . . . . .	110
6.4	Cache influence on ESBMC . . . . .	111
<b>7</b>	<b>Future Work</b>	<b>113</b>
7.1	Handling branches individually . . . . .	113
7.2	Symbolic Execution and Bounded Model Checking hybrid . . . . .	113
7.3	Propagate Assumptions . . . . .	114
7.4	Increasing cache efficiency . . . . .	114
7.5	Solving modified formulas . . . . .	114
7.6	Persistent Formula Cache and Canonical Form . . . . .	114
7.7	Heuristically matching solution spaces . . . . .	115
<b>Bibliography</b>		<b>117</b>
<b>Appendices</b>		<b>120</b>
<b>A</b>	<b>Experiment Results</b>	<b>121</b>
A.1	Experiment 1 Results . . . . .	122
A.2	Experiment 2 Results . . . . .	125
A.3	Experiment 3 Results . . . . .	128
A.4	Experiment 4 Results . . . . .	131
A.5	Experiment 5 Results . . . . .	134
A.6	Experiment 6 Results . . . . .	137
A.7	Experiment 7 Results . . . . .	140
A.8	Experiment 8 Results . . . . .	143
A.9	Experiment 9 Results . . . . .	146
A.10	Experiment 10 Results . . . . .	149
A.11	Experiment 11 Results . . . . .	152
A.12	Experiment 12 Results . . . . .	155
A.13	Experiment 13 Results . . . . .	158
A.14	Experiment 14 Results . . . . .	161
A.15	Experiment 15 Results . . . . .	164
A.16	Experiment 16 Results . . . . .	167
A.17	Experiment 17 Results . . . . .	170
A.18	Experiment 18 Results . . . . .	173
A.19	Experiment 19 Results . . . . .	176
A.20	Experiment 20 Results . . . . .	179
A.21	Experiment 21 Results . . . . .	182
A.22	Experiment 22 Results . . . . .	185

# List of Figures

2.1	All paths and path conditions for code in Listing 2.1 . . . . .	4
3.1	Process of converting ESBMC generated SSA Steps to be added to an SMT Solver . . . . .	18
3.2	Formula Representation . . . . .	20
3.3	Formula representation with modification to lookup formulas . . . . .	21
3.4	Formula Representation with successful unsatisfiable lookup . . . . .	21
3.5	Point of interception of formulas before they are sent to the SMT solver	23
3.6	Subset Matching process with regards to Filter operations . . . . .	28
5.1	Performance of all strategies where the baseline had a solving time between 0s and 1000s for data set 1 . . . . .	57
5.2	Performance of all strategies where the baseline had a solving time between 0ms and 10ms for data set 1 . . . . .	58
5.3	Performance of all strategies where the baseline had a solving time between 11ms and 100ms for data set 1 . . . . .	59
5.4	Performance of all strategies where the baseline had a solving time between 101ms and 1000ms for data set 1 . . . . .	60
5.5	Performance of all strategies where the baseline had a solving time between 1s and 10s for data set 1 . . . . .	61
5.6	Performance of all strategies where the baseline had a solving time between 10s and 100s for data set 1 . . . . .	62
5.7	Performance of all strategies where the baseline had a solving time between 100s and 1000s for data set 1 . . . . .	63
5.8	Performance of strategy 1, 2 and 3 for all samples in data set 1 . . . . .	64
5.9	Performance of strategy 3, 4, 5 and 6 for all samples in data set 1 . . . . .	67
5.10	Performance of strategy 4, 7 and 8 for all samples in data set 1 . . . . .	69
5.11	Performance of strategy 4, 22, 7 and 9 for all samples in data set 1 . . . . .	71
5.12	Performance of strategy 10, 11, 12 and 13 for all samples in data set 1 . . . . .	72
5.13	Performance of strategy 13, 14, 15 and 16 for all samples in data set 1 . . . . .	74
5.14	Performance of strategy 17, 18 and 19 for all samples in data set 1 . . . . .	76
5.15	Performance of strategy 20 and 21 for all samples in data set 1 . . . . .	78
5.16	Performance of strategy 20 across all categories for all samples in data set 1	84
5.17	Performance of strategy 22 across all categories for all samples in data set 1	84
5.18	Performance of all strategies where the baseline had a solving time between 0s and 1000s for data set 2 . . . . .	87
5.19	Performance of all strategies where the baseline had a solving time between 0ms and 10ms for data set 2 . . . . .	88

5.20	Performance of all strategies where the baseline had a solving time between 11ms and 100ms for data set 2 . . . . .	89
5.21	Performance of all strategies where the baseline had a solving time between 101ms and 1000ms for data set 2 . . . . .	90
5.22	Performance of all strategies where the baseline had a solving time between 1s and 10s for data set 2 . . . . .	91
5.23	Performance of strategy 20 over all samples in data set 3 . . . . .	94
5.24	Performance of strategy 22 over all samples in data set 3 . . . . .	95
5.25	Performance of strategy 20 across all categories for all samples in data set 3	98
5.26	Performance of strategy 22 across all categories for all samples in data set 3	98
5.27	Performance of strategy 20 over all samples in data set 4 . . . . .	101
5.28	Performance of strategy 22 over all samples in data set 4 . . . . .	102
5.29	Performance of strategy 20 across all categories for all samples in data set 4	105
5.30	Performance of strategy 22 across all categories for all samples in data set 4	105
6.1	Possible paths to be explored with symbolic execution with guards from bounded model checking displayed . . . . .	108
6.2	Possible paths to be explored with symbolic execution with guards to force a return value of -1 . . . . .	109

# List of Tables

2.1	Experimental results for modifying the structure of a formula for Z3 and boolector . . . . .	12
2.2	Experimental results to show effect of simulating a cache hit for Z3 and boolector . . . . .	13
5.1	Overview of strategies and their relevant options . . . . .	55
5.2	Performance of all strategies where the baseline had a solving time between 0s and 1000s for data set 1 . . . . .	57
5.3	Performance of all strategies where the baseline had a solving time between 0ms and 10ms for data set 1 . . . . .	58
5.4	Performance of all strategies where the baseline had a solving time between 11ms and 100ms for data set 1 . . . . .	59
5.5	Performance of all strategies where the baseline had a solving time between 101ms and 1000ms for data set 1 . . . . .	60
5.6	Performance of all strategies where the baseline had a solving time between 1s and 10s for data set 1 . . . . .	61
5.7	Performance of all strategies where the baseline had a solving time between 10s and 100s for data set 1 . . . . .	62
5.8	Performance of all strategies where the baseline had a solving time between 100s and 1000s for data set 1 . . . . .	63
5.9	Performance of strategy 1, 2 and 3 for all samples in data set 1 . . . . .	64
5.10	Performance of strategy 3, 4, 5 and 6 for all samples in data set 1 . . . . .	67
5.11	Performance of strategy 4, 7 and 8 for all samples in data set 1 . . . . .	69
5.12	Performance of strategy 4, 22, 7 and 9 for all samples in data set 1 . . . . .	71
5.13	Performance of strategy 10, 11, 12 and 13 for all samples in data set 1 . . . . .	72
5.14	Performance of strategy 13, 14, 15 and 16 for all samples in data set 1 . . . . .	74
5.15	Performance of strategy 17, 18 and 19 for all samples in data set 1 . . . . .	76
5.16	Performance of strategy 20 and 21 for all samples in data set 1 . . . . .	78
5.17	Identifying overhead of strategy 20 and 22 for data set 1 . . . . .	80
5.18	Performance of strategy 20 across all categories for all samples in data set 1	82
5.19	Performance of strategy 22 across all categories for all samples in data set 1	83
5.20	Performance of strategy 20 across good categories for all samples in data set 1 . . . . .	85
5.21	Performance of strategy 22 across good categories for all samples in data set 1 . . . . .	85
5.22	Performance of all strategies where the baseline had a solving time between 0s and 1000s for data set 2 . . . . .	87

5.23	Performance of all strategies where the baseline had a solving time between 0ms and 10ms for data set 2 . . . . .	88
5.24	Performance of all strategies where the baseline had a solving time between 11ms and 100ms for data set 2 . . . . .	89
5.25	Performance of all strategies where the baseline had a solving time between 101ms and 1000ms for data set 2 . . . . .	90
5.26	Performance of all strategies where the baseline had a solving time between 1s and 10s for data set 2 . . . . .	91
5.27	Performance of all strategies over all samples in data set 3 . . . . .	93
5.28	Performance of strategy 20 over all samples in data set 3 . . . . .	94
5.29	Performance of strategy 22 over all samples in data set 3 . . . . .	94
5.30	Performance of strategy 20 across all categories for all samples in data set 3	96
5.31	Performance of strategy 22 across all categories for all samples in data set 3	97
5.32	Performance of all strategies over all samples in data set 4 . . . . .	100
5.33	Performance of strategy 20 over all samples in data set 4 . . . . .	101
5.34	Performance of strategy 22 over all samples in data set 4 . . . . .	101
5.35	Performance of strategy 20 across all categories for all samples in data set 4	103
5.36	Performance of strategy 22 across all categories for all samples in data set 4	104
6.1	Performance of strategy 20 and 22 across data set 1 and data set 2 . . .	110

# Listings

2.1	C code example from Green . . . . .	4
2.2	C code for BMC example . . . . .	6
2.3	Unwind Bound k = 1 for Listing 2.2 . . . . .	6
2.4	SMT equivalent for Listing 2.3 . . . . .	6
2.5	Unwind Bound k 2 for Listing 2.2 . . . . .	7
2.6	SMT equivalent for Listing 2.5 . . . . .	7
3.1	Pseudo Code for approach 1 of Range Filter . . . . .	30
3.2	Pseudo Code for approach 2 of Range Filter . . . . .	30
3.3	Pseudo Code for Shape Filter . . . . .	30
3.4	Pseudo Code for Range Filter . . . . .	31
4.1	Pseudo Code for Variable Renaming . . . . .	42
4.2	Pseudo Code for Finding Variable Dependent formulas . . . . .	44

# Acronyms

**BMC** Bounded Model Checking.

**CT** Cache Time.

**FOL** First-Order Logic.

**HR** Hit Rate.

**LT** Baseline Solver Time Range.

**LT** Lookup Time.

**QF** Quantifier Free.

**Sat** Satisfiable.

**SMT** Satisfiability Modulo Theories.

**SSA** Single Static Assignment.

**ST** Solver Time.

**TT** Total Time.

**Unsat** Unsatisfiable.

**VRT** Variable Renaming Time.

**x** Speedup.

# Glossary

**Baseline Solver Time Range** The minimum and maximum amount of time it took for the baseline strategy to return a satisfiability result.

**Cache Time** The amount of time it takes to store a formula in the cache.

**Exact Lookup** Denoted as  $\text{exact-lookup}(f)$ .

**Exact Store** Denoted as  $\text{exact-store}(f)$ .

**Experiment** The results obtained from a corresponding *Strategy* for a given data set, used interchangeably with *Strategy*.

**First-Order Logic** Builds on propositional logic by introducing predicates and quantification.

**Hit Rate** Hit Rate is a measurement of how many times the cache was able successfully retrieve a formula during an iteration.

**Lookup** Denoted as  $\text{lookup}(f)$  which refers to the lookup of a formula  $f$  through any of the three available relevant lookup processes namely Exact Lookup, Subset Lookup andor Unsat Core Lookup.

**Lookup Time** The amount of time it takes to store a formula in the cache.

**Path** A sequence of state transitions through a control flow graph or state transition graph.

**Path Constraint** A branch found along a path that forces a given variable to be limited in the values it is allowed to be assigned for it to be able to transition further along the path.

**Sample** The result obtained from computing the satisfiability of a program for a specific unwind bound  $k$  with ESBMC.

**Satisfiable** Given a propositional formula  $f$ , there exists an assignment to each variable such that the terms in  $f$  evaluate to *True*.

**Single Static Assignment** Static Single Assignment is an intermediary form that is used by compilers for optimizations where any variable can only be assigned a value once [20]. Subsequent modification to the variable would create a new variable to reflect the modification.

**Solver Time** The amount of time spent by the SMT solver to produce a satisfiability result.

**Speedup** A ratio between two values to indicate an increase in performance (which could be a decrease in time or increase in for some other metric) or decrease in performance where a value greater than 1.0 constitutes an increase.

**Store** Denoted as  $store(f)$  which refers to the storing of a formula  $f$  through any of the three available storing processes namely Exact Store, Subset Store or Unsat Core Store.

**Strategy** A predefined collection of options that will determine how the cache will operate, used interchangeably with *Experiment*.

**Subset Lookup** Denoted as  $subset\text{-}lookup(f)$ .

**Subset Store** Denoted as  $subset\text{-}store(f)$ .

**Total Time** The total amount of time it takes ESBMC to obtain a satisfiability result for a single unwinding of unwind bound  $k$  during a verification run.

**Unsat Core Lookup** Unsatisfiable Core Lookup denoted as  $unsat\text{-}core\text{-}lookup(f)$ .

**Unsat Core Store** Unsatisfiable Core Store, denoted as  $unsat\text{-}core\text{-}store(f)$ .

**Unsatisfiable** Given a propositional formula  $f$ , there exists no assignment to each variable such that the terms in  $f$  evaluate to *True*.

**Variable Renaming Time** The amount of time it takes to rename all variables in a formula  $f$  into a standardized form.

**Verification** The process of establishing the accuracy of a given system.

**Verification Run** The process of verifying a single ANSCI-C program for any property violations.



# Chapter 1

## Introduction

Techniques for automatic verification of both software and hardware systems have developed at a rapid pace since 1982 when *model checking* [3, 17, 14, 15] was described by Clarke and Emerson. Model checking consists of a set of techniques and algorithms to express the model to be verified as a finite state machine. Temporal logics can then be used to reason about properties and events in time without explicitly introducing the concept of time in the models [27]. Due to the high level of automation offered by model checking, it has been implemented by many hardware and software manufacturers [5] as part of their quality assurance process.

Early implementations of model checking produced explicit representations of the finite state machines through state transition graphs which were explored through effective graph traversal techniques. During the exploration of the state transition graph properties are checked for violations. If a violation is found where a property no longer holds, it is considered that a bug or a counterexample has been identified. The path traversed to the state containing the violation can be used to produce a set of assignments to reach said state. Finding a property violation relies on a system specification to describe what constitutes a property violation. Due to this contractual agreement with the system specification, it is impossible to determine whether a system is correct, but rather only that it conformed to the supplied system specification. Since model checking makes use of explicit state transition graphs the distinction is made that model checking is used to falsify properties rather than to verify them since it cannot show that the property holds for state transitions not represented in the current state transition graph. This technique was successfully applied to the verification of hardware systems but found limited success for software verification with only small programs being checked for property violations within a reasonable amount of time. These small examples were already represented by millions of nodes in their respective state transition graphs. When industrial size software systems were introduced the techniques faltered as the state transition graphs needed to represent these systems grew exponentially while the resources available at the time were not able to handle the demands. The term *state space explosion* [16] was coined to describe this inherent problem when making use of explicit state transition graphs.

In 1991 a new technique was suggested by Coudert et al. namely *symbolic state space exploration* [10, 19] also referred to as symbolic model checking<sup>1</sup>. Symbolic model checking did away with explicit state representation by making use of BDDs

---

<sup>1</sup>Symbolic execution [23] was described by King in 1976

(Binary Decision Diagrams), replacing concrete variable assignments with symbolic assignments. The introduction of symbolic execution allowed for software systems with more than  $10^{20}$  explicit states [10] to be reasoned about. Although this was a factor of magnitude larger than what could be achieved through model checking, it was still only capable of handling small components with regards to industrial size software systems.

In 1999 Biere et al. described a new model checking technique called bounded model checking [7] which replaced BDDs as used in symbolic execution with propositional formulas to be solved by a SAT solver. The technique aims to verify that a state or set of states can be reached where the verification includes identifying loops in state transition graphs. The states that are checked for reachability represent properties to be checked for violations along a path that is bound in length by a number  $k$  called the unwind bound. These states and paths are encoded as propositional formulas in SSA form to be checked for satisfiability by SAT solvers which seldom requires exponential space compared to the BDDs used by symbolic execution which often does. The unwind bound is then increased incrementally until a property violation is found (a satisfiable assignment from the SAT solver) until the completeness threshold is reached or until the verification procedure is stopped as the technique cannot show that a property always holds for paths of infinite length. The paths of length  $k$  are explored by using a strategy called *iterative deepening*, also known as *iterative deepening depth-first search*. The technique is used for state transition graphs and limits the depth of the paths explored to the unwind bound  $k$ . It uses less space than a breadth-first search but produces the same results due to it being bound in depth.

The process of incrementally increasing the unwind bound  $k$  for bounded model checkers while generating states to be checked for reachability through iterative deepening could create a situation where a state is generated along with a sequence of transitions that have been seen during previous iterations. These states and transitions are then encoded into propositional formulas to be sent to a SAT solver. Recalculating the reachability of these states with the help of the SAT solver constitutes a great amount of time being spent on a problem for which the results were already computed during previous iterations.

A classic technique used in computer science to improve the performance of a system is to trade off time for space (or vice versa in some cases). If a result can be stored and looked up faster than it would have taken to compute the result (within a reasonable amount of space), then it is a worthwhile to rather store it. The limitations present on the increase in performance of SAT solvers due to problems being NP-complete along with the possibility that Bounded Model Checkers requires the same states to be checked for reachability in subsequent unwindings, makes it an attractive candidate for the technique of trading time for space. It has already been shown that the caching of results achieved a performance increase in the field of symbolic execution [29], implying further that the possibility to achieve an increase in performance can be obtained in the field of bounded model checking by implementing similar caching techniques.

This document will describe the design and evaluation of a bounded model checking cache to be implemented into the ESBMC framework. Techniques and strategies from symbolic model checking will be replicated along with some novel work to show that caching of formulas to be checked for satisfiability by a SAT solver can be stored and reused to increase the performance of ESBMC and the overall potential of caching in the case of no net increase in performance.

# Chapter 2

# Background

## 2.1 Model Checking

*Software Model Checking* encompasses a set of techniques and algorithms to automatically extract a model from a piece of code (model here does not refer to an abstraction of the code [13]) that is then exhaustively checked against a specification for any violations. Originally verification of software was done by handwriting proofs using a Floyd-Hoare style logic, the best known being proposed by Owicki and Gries [26, 13]. This process quickly became cumbersome as software grew in size and complexity, an alternative method was needed that assured accuracy and was computable within a reasonable amount of time. *Model Checking* enjoyed success from an early stage when applied to the verification of hardware systems. Hardware systems could quite easily be represented as a finite state transition graph (with some exceptions) and then verified against a specification. Software Systems, on the other hand, suffered from the *state space explosion problem* [16] and had complex data structures and arithmetic that could potentially create infinite state spaces when undefined loop lengths and recursion come into play. A key problem within the verification of software systems stems from concurrent programs as the number of possible interleavings exacerbates the existing problem of *state space explosion*. *Software Model Checking* has evolved along different approaches to handle these problems, namely *Explicit State Model Checking*, *CEGAR Based Model Checking*, symbolic model checking and bounded model checking or in some cases combinations of these. *CEGAR Based Model Checking* (Counterexample-guided abstraction refinement [11]) attempts to use traditional *Model Checking* approaches of large finite state transition graphs by generating an abstract representation of the initial transition graph by means of overestimation. Every path through the new transition graph can be related to a path in the original transition graph. The abstract state transition graph can then be checked through traditional methods. If the abstract transition graph is found to not be correct, then a counterexample generated can be used to test against the original state transition graph. If this counterexample from the abstraction is still a counterexample of the original model, then the original system is not correct. On the other hand, if the counterexample did not disprove the correctness of the original graph, then the abstract representation is refined and the process started again. symbolic model checking and bounded model checking will be discussed in greater detail in Section 2.2 and 2.3 as key comparisons to caching will be discussed later on in this document.

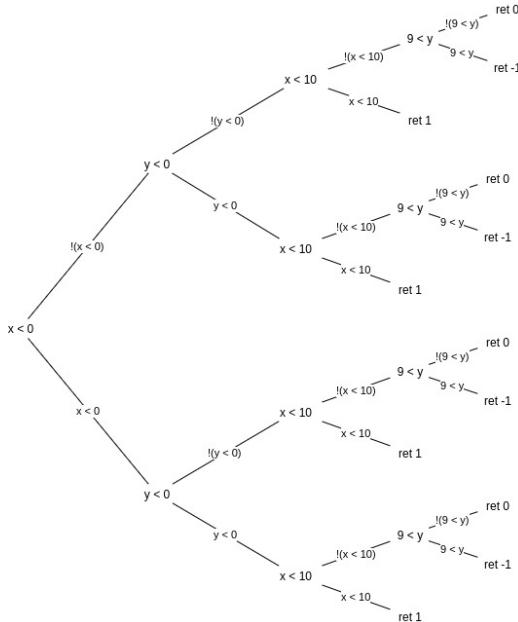
## 2.2 Symbolic Model Checking

Symbolic Model Checking allows for reasoning about multiple states at the same time. It achieved a significant increase in the number of states that could be explored in a state transition graph without explicitly creating all the states, but rather grouping some states and their transitions. Symbolic execution works on the premise that a variable can be assigned all possible values (within the bounds of the data type) when created. The state transition graph is then explored through a breadth-first search procedure. Each time a branch condition is found, the state representing the condition being checked by the branch condition is split in two by diving the currently represented states into a group that would fail the branch condition and another group that would pass the branch condition. These groups are only abstract concepts while they are actually being represented by path conditions along the way. Consider the following example taken from Green [29]:

```

1 int m(int x, int y) {
2     if (x < 0) x = -x;
3     if (y < 0) y = -y;
4     if (x < 10) return 1;
5     else if (9 < y) return -1;
6     else return 0;
7 }
```

**Listing 2.1:** C code example from Green



**Figure 2.1:** All paths and path conditions for code in Listing 2.1

When the code in Listing 2.1 is checked by a *Symbolic Execution Tool*, the variables  $x$  and  $y$  can be assigned any value within the integer range as there have been no path

conditions restricting them along the initially empty path  $p = \emptyset$ . As soon as the branch condition in line 2 is reached, the values that can be assigned to  $x$  is split into two separate groups, those where the value of  $x$  is less than 0 and those where  $x$  is greater or equal to 0. These conditions are added to the path  $p$  to produce two new paths to be checked namely  $p_0$  and  $p_1$ .

$$p_0 = x < 0 \quad (2.1)$$

$$p_1 = x \geq 0 \quad (2.2)$$

The code is then explored further with regard to  $p_0$  and  $p_1$ . The next branch condition is encountered in line 3 and two new paths are available for each of the currently known paths resulting in a total of 4 new paths namely  $p_2$  through  $p_5$ .

$$p_2 = x < 0 \wedge y < 0 \quad (2.3)$$

$$p_3 = x < 0 \wedge y \geq 0 \quad (2.4)$$

$$p_4 = x \geq 0 \wedge y < 0 \quad (2.5)$$

$$p_5 = x \geq 0 \wedge y \geq 0 \quad (2.6)$$

It is easy to see that the number of paths that can be explored in a program grows exponentially in size for each branch condition that is found.

Each time a new path condition is added to a path it needs to be checked for satisfiability. If the formula  $p$  is not satisfiable, then the path that follows it cannot be reached with the current path constraints. If the path that follows can never be reached for any combination of path constraints then a piece of dead code has been found (no coverage possible). This path no longer needs to be explored further at that point. If the propositional formula was satisfiable then that path can be explored further during subsequent steps from the breadth-first search. This process will continue until all paths have been explored or until a property violation was found.

Listing 2.1 contains 5 branches implicitly, if all paths were to be checked there would be a total of 12 paths by the end and a total of 24 queries to be sent to the SMT solver. If all paths are checked as soon as they are constructed and unsatisfiable paths no longer explored, the total of queries sent to the SMT solver will decrease to 22.

## 2.3 Bounded Model Checking

Bounded Model Checking originally came about when SAT solvers started to gain popularity with their rapid increase of reasoning power. SAT solvers were quickly able to handle instances with millions of clauses and hundreds of thousands of variables [8]. The technique of bounded model checking assumes that a property violation will occur within a given number of steps  $k$  where the violation occurred along any possible path that is reachable within those  $k$  steps. If no property violation is found, the unwind bound  $k$  is increased by one and the process started again. Since the unwind bound  $k$  is increased incrementally, it mimics the behavior of a breadth-first search over all possible program paths. The added benefit of this approach is if there is indeed a counterexample or a path for which a property violation is found, then it is also the shortest path to that property violation, as any shorter path would have already been discovered with the violation reported. The process of bounded model checking moves away from verification and instead focuses on falsification. It is not possible for bounded model checking to show that the property violation never occurs, it can only show when a property violation occurs within  $k$  steps.

## Example

In the following example, a conceptual overview will be given of how a file containing valid C code gets checked for satisfiability by using bounded model checking.

```

1 int main() {
2     int x = 1;
3     int y = 2;
4
5     for (int z = 0; z < 1; z++) {
6         assert(x != y);
7         x += 1;
8     }
9 }
```

**Listing 2.2:** C code for BMC example

The code listed is a very simple for loop that asserts that the value of  $x$  is not equal to the value of  $y$ . Each execution of the for loop will increase the value of  $x$  by 1 with the assert failing during the second iteration of the loop. The loop only executes once though due to its bounds, so no violation is possible for the listed code.

Conceptually the code found within a loop is duplicated a number of times equal to the unwind bound  $k$ , with a bound check at the start of each piece of code to determine if the code should be executed. The following listing shows this conceptual view of the code for the unwind bound  $k = 1$ .

**Listing 2.3:** Unwind Bound  $k = 1$  for Listing 2.2

```

x = 1;
y = 2;
z = 0;

if (z < 1) {
    assert(x != y);
    x1 = x + 1;
    z1 = z + 1;
}
```

**Listing 2.4:** SMT equivalent for Listing 2.3

```

(= x 1)
(= y 2)
(= z 0)
(= guard1 (< z 1))
(= x1 (+ x 1))
(= z1 (+ z 1))
(and guard1 not(= x y))
```

The SMT formula listed in Listing 2.4 shows how operations performed on variables are encoded in *SSA Form*. When the value of  $x$  and  $z$  are incremented they are assigned new variables instead of modifying the original. This allows the SMT solver to keep track of when and where variables are modified, especially if there are different paths that a variable will be modified along.

The code is converted into a state transition system before being converted again into a boolean formula to represent a sequence of state transitions to reach a state of interest. In this case, the state of interest would be the state that checks the validity of the `assert` statement. With the goal of finding a property violation, the SMT solver needs to attempt to find a sequence of states that would violate the `assert` check. To achieve this, the negation of the `assert` statement is used during the conversion to the boolean formula. Listing 2.4 shows the FOL formula created to represent the state

transition system and the state of interest to be checked for satisfiability by the SMT solver.

Here the idea of a guard is introduced where the `if` statement was encountered. Guards represent any point in the transition system where a branch is encountered with one of two paths forward. When the code specifies more than two possible paths (for example an `if`, `elif`, or `else` statement), the paths are converted into multiple `if` `else` paths. For the `assert` to be violated, the bound condition for the branch has to hold. The formula reflects this by specifying that both the `guard` and the negated `assert` has to evaluate to true.

For bound  $k = 1$  the SMT solver will return an unsatisfiable result since there is no assignment of the variables to satisfy the formula.

Listing 2.2 shows this conceptual view of the code for the bound  $k = 2$ . The code is converted into transition system and subsequently into the formula in Listing 2.6.

**Listing 2.5:** Unwind Bound k 2 for

**Listing 2.2**

```
x = 1;
y = 2;
z = 0;

if (z < 1) {
    assert(x != y);
    x1 = x + 1;
    z1 = z + 1;
}
if (z1 < 1) {
    assert(x1 != y);
    x2 = x1 + 1;
    z2 = z1 + 1;
}
```

**Listing 2.6:** SMT equivalent for Listing 2.5

```
(= x 1)
(= y 2)
(= z 0)

(= guard1 (< z 1))
(= x1 (+ x 1))
(= z1 (+ z 1))
(= guard2 (< z1 1))
(= x2 (+ x1 1))
(= z2 (+ z1 1))

(or
(and guard1 not(!= x y))
(and guard2 not(!= x1 y)))
```

A slight change in structure appears compared the previously generated formula, as the `assert` statements now appear within a disjunct at the end of the formula. The goal of the program verification is to find a sequence of states to a state of interest (a property violation), since there are now multiple states of interest the path to any of these states is sufficient instead of all paths needing to appear in the same sequence.

Again no satisfiable assignment could be found for the stated formula, even though there is a clear `assert` violation, the `guard` checking the bounds of the `if` statement to reach the `assert` could not be satisfied.

For this example, the bound  $k$  would be increased infinitely<sup>1</sup> trying to find a property violation where none is possible. This is an example of how bounded model checking is very useful for proving that a property violation exists at a specific bound  $k$ , but not that no property violation can ever exist.

---

<sup>1</sup>Assuming the completeness threshold [8] is not known

## Bounded Model Checking Tools

Although there are plenty of bounded model checking tools, only two will be described shortly namely *CBMC* and *ESBMC* as the latter will be used for the integration of a formula cache.

### **CBMC**

*CBMC* [12] is a tool created in 2004 by Clarke et al. that formally verifies ANSI-C programs using bounded model checking. The tool is capable of handling all operators and pointer constructs defined in ANSI-C. Properties that can be checked for include pointer safety, array bounds, and user-provided assertions.

### **ESBMC**

*ESBMC* [18] is a branch of *CBMC* designed to address some of the shortcomings of *CBMC* with regards to the increase in size of the generated propositional formula as well as the loss of structure during translation. To achieve this, theories that are richer than propositional logic is used in conjunction with SMT solvers instead of SAT solvers.

During the document, there will be an occasional heading *ESBMC Program Options* that will specify options that can be enabled for *ESBMC* that will dictate how formulas are modified as well as how the cache will behave.

## 2.4 Existing Formula Caches

Symbolic Execution has already been proven to benefit from using formula caches. Caches that have already been shown to improve the overall performance for symbolic execution techniques include Green [29], GreenTrie [22], Utopia [2], Recal [1] and Recal+ [1]. The approaches from Green will be analyzed in more detail to guide the design of the *Bounded Model Checking Cache* although other caches should be explored in the future in case there are other techniques that can be carried over as well. Green is chosen as it only follows a few basic steps to modify the formula before it is stored or retrieved.

Green is a persistent formula cache specifically built for use with symbolic execution techniques. The formulas are stored in memory with redis, allowing for fast storage and lookups as well as having the added benefits of being persistent and that the cache can be accessed easily from other programs that can read the redis store or that interact through the Green interface. Green only makes use of four basic operations during its processing of formulas namely slicing, canonizing, lookup and store.

### Slicing

When a new constraint  $c$  is introduced along a path  $p$  of which a path is a conjunct of constraints already, only part of the constraints along the path might be required. If  $p$  is currently unsatisfiable then introducing another constraint will not change the satisfiability. If the path is currently satisfiable, then the new constraint  $c$  could change the satisfiability. To decrease the size of the formula to be checked (potentially saving computation time as well as increasing likelihood of reuse) a reachability graph is used from the variables in  $c$  to the variables along the path  $p$ . The constraints forming part of  $p$  for which no variables were reachable can be removed (sliced) to form a new path  $p'$ .<sup>2</sup> In short, this process identifies all constraints in the conjunct that are influenced by the new constraint and builds a new path from them. Path  $p'$  is then passed on to the Canonization step.

### Canonization

The main procedure for canonization would normally be to transform the path conditions into the normal form for linear arithmetic. Green implements two additional modifications to increase the likelihood of finding a cache hit. Pre-Heuristic which is a weak reordering that reorders all path conditions according to the lexicographical ordering of the variables within each constraint. This step is followed by the standard transformation into normal form for linear arithmetic. Post-Heuristic is the final part of the reordering which again reorders the constraints according to the lexicographical order followed by a renaming of all variables into standardized names. The new formula is then sent to the Lookup and Store step<sup>3</sup>.

### Lookup and Store

After the formula has been canonized, it is looked up in the cache to determine if the formula has been checked for satisfiability before. If the formula is found during the lookup process, the satisfiability result is returned and the next path can be explored. If there exists no match in the cache, the formula is translated into the relevant solver and the result computed. The formula supplied by the canonization step then gets stored along with the satisfiability result from the solver is stored in the cache. Finally, the satisfiability result is return and the next path can be explored<sup>4</sup>.

Referring back to the code found in Listing 2.1, the paths along with their satisfiability results can now be stored for potential reuse. There are now a total of 6 queries sent to the SMT solver compared to the previously stated 22 as the remaining 16 were already found in the cache.

Most of the paths explored during symbolic execution are expected to find path conditions that allow for more paths to be explored with only a few path conditions not allowing for the path to be explored further. This creates a cache where the formulas within it are mostly satisfiable with a much smaller amount of formulas listed as unsatisfiable. Green saves symbolic execution time by avoiding a solver call altogether when a formula is found in the cache, this is slightly different from the procedure that

---

<sup>2</sup>See section 4.2 for the equivalent implementation for the Bounded Model Checking Cache.

<sup>3</sup>See section 4.4

<sup>4</sup>See section 3.6 and 3.7

needs to be followed for the implementation of a formula cache in a Bounded Model Checking tool, as will be discussed later.

## 2.5 SMT Solvers

Satisfiable Module Theories [6, 4] (SMT) Solvers take a quantifier-free First Order Logic formula along with a combination of theories and determines if there are assignments to the free variables such that the formula evaluates to true. The exact structure and content of these formulas are entirely dependent on the theories that are selected along with them. These theories allow for formulas to be much richer compared to propositional formulas as used by SAT solvers. For the purpose of the evaluation in this document, only two SMT solvers will be discussed and used during experiments, namely boolector and Z3.

### 2.5.1 boolector

boolector [9, 25] is an efficient SMT solver for *Quantifier-free theories* of fixed-size bit vectors and array developed at the *Johannes Kepler University*. The solver makes use of techniques like *term rewriting*, *bit blasting*, *lemmas on-demand* [28] and *don't care reasoning* [24]

### 2.5.2 Z3

Z3 [21] is an SMT solver created by *Microsoft Research* and uses a list of novel techniques like *Quantifier Instantiation* and *Theory Combination* while determining the satisfiability of a given formula.

### 2.5.3 Solver Unpredictability

SMT solvers take a QF FOL formula and attempts to assign values to variables such that the formula evaluates to True. The SMT solvers employ an array of strategies to find an assignment as quickly as possible. A basic explanation of the approaches can be summed up as follows:

- Decide - Take a variable with no assigned value and assign it a value.
- Deduce - Given the new variable assignment, deduce possible values for other variables with no assignment.
- Resolve - In case a conflict now arises, undo the assignments from the previous steps and attempt a different assignment.

Modern SMT solvers employ a range of techniques and heuristics to help guide these processes. Consider a simple SMT solver with no heuristics or capabilities of inferring information from the different parts of the formula. The time it would take assign values to variables can vary greatly depending on the order the variables are assigned values, as a wrong decision can cause many conflicts that would need to be resolved.

Given the following example, assume the naive approach where variables will be assigned values incrementally in the order of which they are encountered where no

information is carried over during the resolution step until a satisfiable result is found or all possible combinations are shown to be unsatisfiable.

```

0 < z < 1000
0 < x < 3
x < z
0 < y < x

z = 1    z = 1    z = 2    z = 2    z = 3    z = 3
x = 1    x = 2    x = 1    x = 2    x = 1    x = 2
unsat    unsat    unsat    unsat    unsat    sat

```

It has taken 6 iterations of variable assignments before a satisfiable result has been found. Given the exact same formula, swap the first two constraints ( $z$  and  $x$ ) and calculate the number of iterations it takes to find the satisfiable assignment.

```

0 < x < 3
0 < z < 1000
x < z
0 < y < x

x = 1    x = 1    x = 1    x = 1    x = 2    x = 2    x = 2
z = 1    z = 2    z = 3    ...   z = 999   x = 2    x = 2    z = 3
unsat    unsat    unsat    unsat    unsat    unsat    unsat    sat

```

It now took 1002 iterations to obtain the satisfiable assignment. Although this is in no way the true reflection of how the SMT solvers go about finding satisfiable or unsatisfiable results, it does show that the order of exploration is important. SMT solvers employ a range of strategies and heuristics to focus on paths that should quickly yield an absolute result. However, it does not guarantee that it will always explore the best path first.

### Replicating unpredictability in modern SMT Solvers

To demonstrate that the behavior described by the naive approach can be replicated in state of the art SMT solvers, two experiments namely *Swap Variable Assignments* and *Reverse Asserts* will be performed on both Z3 and boolector. In these experiments the formulas will be modified into a polymorphic form after which it will be sent to the respective SMT solver and the satisfiability calculated. The time taken to compute the satisfiability of both the unmodified and modified formulas will then be compared to evaluate any differences in computation time.

### Swap Variable Assignments

Given the formula  $f$ , with a term  $t_i$  where the term is an assignment of two variables  $a, b$  such that  $t_i = (a = b)$ , swap the order of the variables such that  $t'_i = (b = a)$  and construct a new formula  $f'$ .

$$f = t_0 \wedge t_1 \wedge \dots \wedge t_{n-1} \quad (2.7)$$

$$t_i = (a_i = b_i) \quad (2.8)$$

$$t'_i = (b_i = a_i) \quad (2.9)$$

$$f' = t'_0 \wedge t'_1 \wedge \dots \wedge t'_{n-1} \quad (2.10)$$

### Reverse Asserts

Given the formula  $f$  where the last term  $t_{n-1}$  is a disjunct of asserts (represented by terms), reverse the order of the terms and create a new formula  $f'$  such that:

$$f = t_0 \wedge t_1 \wedge \dots \wedge t_{n-1} \quad (2.11)$$

$$t_{n-1} = a_0 \vee a_1 \vee \dots \vee a_{m-1} \quad (2.12)$$

$$t'_{n-1} = a_{m-1} \vee a_{m-2} \vee \dots \vee a_0 \quad (2.13)$$

$$f' = t_0 \wedge t_1 \wedge \dots \wedge t_{n-2} \wedge t'_{n-1} \quad (2.14)$$

### Comparisons of modifications

To evaluate the effect on the solving time of the previously stated formula modifications, the following experiment will be set up. ESBMC will be used to generate the formulas with an unwind bound of  $k = 1$ . The file to be parsed by ESBMC will be ‘rekh\_nxt\_true-unreach-call.1.M1.c’ from the ‘seq-mthreaded’ group that can be found in the SV-COMP 2016 data set. To ensure an accurate comparison each specific formula paired with a specific solver will be run five times and the average time used. The results for the baseline (unmodified formula) and the modified formulas are shown in the following table:

settings	boolector time(ms)	z3 time(ms)
baseline	4554	3524
reverse asserts	4531	2343
swap assignment order	4514	5060
reverse asserts and swap assignment order	4536	5359

**Table 2.1:** Experimental results for modifying the structure of a formula for Z3 and boolector

While using boolector as the SMT solver there seems to be no noticeable variation in the time it takes to solve the formula, Z3, on the other hand, varies greatly with a decrease of 1.2s and an increase of 1.8s for different settings, implying the modification may cause more volatile timings with regard to solving time.

### Removing Asserts

Similarly to the variation in timings found from modifying formulas, solving times can also be influenced by decreasing the size of a formula. Formulas constructed by ESBMC (See section 3.2) contain a disjunct of asserts as part of the final term in the conjunct. It could be expected that decreasing the size of this disjunct would result in less time taken by the SMT solver to compute a satisfiable result. To evaluate the effect of decreasing the size of the formula by removing one of these asserts, the following experiment will be run. The same setup as the previous experiment will be used but the formula modification will now be in the form of removing an assert from the index  $i$  in the disjunct of asserts.

Remove Assert	boolector time (ms)	Z3 time (ms)
-	4554	3254
0	4560	2914
1	4542	3861
2	4554	2412
3	43079	5438
4	6799	4510
5	4400	3708
6	4479	4894
7	4450	4665
8	4392	4012
9	4524	3230
10	4994	4295

**Table 2.2:** Experimental results to show effect of simulating a cache hit for Z3 and boolector

The same trend follows where boolector is mostly consistent except for removing the *assert* at position 4, causing a bit of a slow down and the removing of the *assert* at position 3 which has a drastically longer solving time. Similar to the previous experiment, Z3 the times taken to compute the satisfiability vary greatly between a decrease of 0.8s and an increase of 2.2s, even though the formula decreased in size compared to the baseline.

### Conclusion

It is clear to see that altering the structure of the formula without altering the satisfiability or set of possible assignments of this formula, can greatly affect the time to solve said formula even with a modern SMT solver. These results are seen in many different examples, showing it is not just an isolated case, although the degree in which the timings change vary from experiment to experiment (see Section 5). A formula cache that is specifically implemented for bounded model checking will be altering a formula by slicing some *assert* statements that could cause unpredictable consequences regarding the amount of time it takes for the SMT solver to compute the satisfiability. This needs to be taken into account during Section 5.

### **ESBMC Program Options**

```
--swap-assignment-order  
--reverse-asserts
```

# Chapter 3

# Cache Design

This chapter will describe the integration of a formula cache into ESBMC, along with overall design and strategies to be implemented as to achieve efficient and fast storing and retrieval of formulas. Goals will be set out to describe what would constitute an effective operational cache. During some examples, trivial formulas might be used that is easily simplified or the satisfiability deduced but the cache does not do any reasoning about the content of the formulas or terms with regard to satisfiability, and they are used purely for illustrative purposes.

## 3.1 Goals

### 3.1.1 Decreasing End-To-End Time Of ESBMC

The primary goal of any cache is to store and retrieve results faster than it would take to recompute the results. When a cache is part of a larger program where the influence of the cache cannot be accurately predicted (see Section 2.5.3), the primary goal changes to obtain an increase in overall performance of the program. In terms of bounded model checking and ESBMC in particular, an increase in performance would constitute a decrease in end to end time of the verification of a program.

### 3.1.2 Decreasing Solver Time And Increasing Cache Hits

If the primary goal was not obtained to decrease the total amount of time spent by ESBMC during a verification run, then a good secondary goal would be to decrease the amount of time it takes the SMT solver to calculate the satisfiability of a formula by decreasing the size of the formula to be evaluated. It has already been shown that the performance of the SMT solver could decrease when the formula is altered or decreased in size (Section 2.5.3) so a net increase in performance (net decrease in solving time) will show the potential to have a net increase in performance for ESBMC. If it is shown that the SMT solver does perform better with these modified formulas then any increase in performance of the cache with regards to the amount of time it takes to look up or store a formula would directly decrease the total amount of time it takes ESBMC to do a verification run as the formulas and solving time should stay consistent, creating the opportunity to achieve or move closer to the primary goal.

### 3.1.3 Across-Run Reuse

A tertiary and last goal would be to store and lookup formulas to be used across program verification runs as has been shown to be successful in the field of symbolic execution.

## 3.2 Understanding ESBMC Formulas

To integrate a formula cache into ESBMC, first, it must be known what the structures of the formulas are when they are sent to the SMT solver and secondly it must be determined where to intercept these formulas to allow for storage and retrieval. ESBMC generates a list of SSA Steps to be added to the SMT solver, where each step can be identified as one of three different kinds of steps, namely assignments, assumptions and asserts. Each of the three different steps represents different parts of the ANSI-C code that is being verified in SSA form as follows:

**Definition 3.2.1.** Assignment The assignment of values to variables as well as operations on them. Assignment of variables could also be in the form of the result from a bounds check or successful execution of a method.

**Definition 3.2.2.** Assumption Assumptions are added in the form of an unwinding assumption that assumes the upper bound of a loop was satisfied. Manual assumptions can also be added to narrow the potential values of specific variables by the user.

**Definition 3.2.3.** Assert A property check for property violations, these might include division by zero, array out of bounds etc. These are the states of the program that are of interest as reaching them would cause the verification of the program to fail.

Each assignment, assumption and assert is represented by a term  $t$  which can be defined as a 3-tuple with the following values: An operator  $o$ , a list of child terms  $l$  on which the operator is performed and a variable or value  $v$ .

$$t = (o, l, v) \quad (3.1)$$

$$l = [child_1, child_2, \dots, child_{n-1}] \quad (3.2)$$

The term  $t$  can only have 2 possible configurations as follows: A term  $t$  with a set operator  $o$  defining an operation on the non-empty list of child terms  $l$  and an empty variable or value  $v$ .

$$t = (o, l, \emptyset) \quad (3.3)$$

$$l = [child_1, child_2, \dots, child_{n-1}] \quad (3.4)$$

Or a term  $t$  with no operator  $o$ , an empty list of child terms  $l$  and a variable or value  $v$ .

$$t = (\emptyset, \emptyset, v) \quad (3.5)$$

This creates a recursive structure as the child terms themselves may also contain more operators and their own child terms, resulting in a tree with all the leaf nodes

representing variables or absolute values. These terms create the building blocks of the formulas to be sent to the SMT solver and to be handled by the cache.

When an assignment is encountered, it is added directly to the SMT solver. Assumptions on the other hand are added to a list containing all assumptions seen so far up until that point. When an assert is encountered, a new term  $t$  is built as follows where  $b_0$  through  $b_{n-1}$  represent all the assumptions that have been encountered thus far when the assert appears.

$$h = b_0 \wedge b_1 \wedge \dots \wedge b_{n-1} \quad (3.6)$$

$$t = \neg(h \implies \text{assert}) \quad (3.7)$$

The representation of the term  $t$  with regards to its specified structure would be as follows:

$$t = (\neg, l_0, \emptyset) \quad (3.8)$$

$$l_0 = [t_1] \quad (3.9)$$

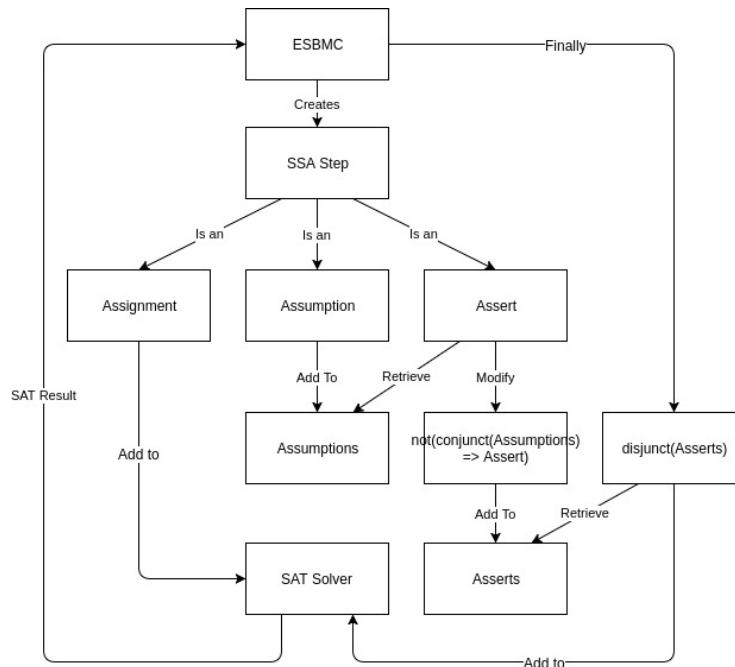
$$t_1 = (\implies, l_1, \emptyset) \quad (3.10)$$

$$l_1 = [h, \text{assert}] \quad (3.11)$$

$$h = (\wedge, l_2, \emptyset) \quad (3.12)$$

$$l_2 = [b_0, b_1, \dots, b_{n-1}] \quad (3.13)$$

The new term  $t$  is then added to a list of asserts. After all the SSA Steps have been processed, the list of asserts is turned into a new term by creating a disjunct of all the elements in the list. This new term is then added to the SMT solver and then checked for satisfiability after which the result is returned back to ESBMC. This process is illustrated in Figure 3.1.

**Figure 3.1:** Process of converting ESBMC generated SSA Steps to be added to an SMT Solver

Consider the following formula  $f$  that represent the equations sent to the SMT solver by ESBMC with the terms  $t_0$  to  $t_{n-2}$  representing the *assignments* and term  $t_{n-1}$  representing the disjunct of *asserts*. This structure will represent the formula expected by the cache to be modified, looked up and eventually stored along with a satisfiability result.

$$f = t_0 \wedge t_1 \wedge \dots \wedge t_{n-1} \quad (3.14)$$

$$t_{n-1} = a_0 \vee a_1 \vee \dots \vee a_{m-1} \quad (3.15)$$

Any formula  $f$  is itself a term as well with the tuple values of:

$$f = (\wedge, l, \emptyset) \quad (3.16)$$

$$l = [t_0, t_1, \dots, t_{n-1}] \quad (3.17)$$

The following notation will be used in subsequent operations to access the specific values of a term  $t$ :

$$O(t) = o \text{ value of term } t \quad (3.18)$$

$$L(t) = l \text{ value of term } t \quad (3.19)$$

$$V(t) = v \text{ value of term } t \quad (3.20)$$

The notation  $L(t)_i$  represents the operation of accessing the list of child elements  $l$  of  $t$  and accessing the child term at position  $i$  in that list.

### 3.3 Handling Asserts Individually

A premise for effective reuse of formulas within a single program verification run over multiple bound unwindings, is that some of the *asserts* that get generated should appear again in subsequent unwindings. Currently, the formula to be sent to the cache is in the following form as specified in Section 3.2 where  $t_0$  to  $t_{n-2}$  are the assignments and the last term  $t_{n-1}$  is a disjunct of asserts.

$$f = t_0 \wedge t_1 \wedge \dots \wedge t_{n-1} \quad (3.21)$$

$$t_{n-1} = a_0 \vee a_1 \vee \dots \vee a_{m-1} \quad (3.22)$$

This formula can be converted into a disjunct of conjuncts<sup>1</sup> to be processed by the cache independently from each other. Let the formula  $g$  represent the conjunct of assignments (terms  $t_0$  to  $t_{n-2}$ ) generated by ESBMC. A new formula  $f'$  can then be constructed by combining  $g$  with each assert  $a_i$  found in the disjunct term  $t_{n-1}$ .

$$f = g \wedge t_{n-1} \quad (3.23)$$

$$t_{n-1} = a_0 \vee a_1 \vee \dots \vee a_{m-1} \quad (3.24)$$

$$g = t_0 \wedge t_1 \wedge \dots \wedge t_{n-2} \quad (3.25)$$

$$f' = (g \wedge a_0) \vee (g \wedge a_1) \vee \dots \vee (g \wedge a_{m-1}) \quad (3.26)$$

Each clause in  $f'$  can now be handled as an individual formula during any processing, lookups or storing during the caching process. If a term in  $f'$  is satisfiable then the whole formula is satisfiable. Alternatively, if a term in  $f'$  is shown to be unsatisfiable that term can be removed from the formula  $f'$  before it is sent to the SMT solver, creating a smaller and potentially simpler formula to evaluate.

The default behavior of the cache is to handle asserts individually, this however can be overridden to handle all asserts as a combined, single assert by enabling the option `combine-asserts`. This option will have the following effect on  $f$  and  $f'$ .

$$f = g \wedge t_{n-1} \quad (3.27)$$

$$t_{n-1} = a_0 \vee a_1 \vee \dots \vee a_{m-1} \quad (3.28)$$

$$a = t_{n-1}g = t_0 \wedge t_1 \wedge \dots \wedge t_{n-2} \quad (3.29)$$

$$f' = (g \wedge a) \quad (3.30)$$

#### ESBMC Program Options

```
--combine-asserts // Treats disjunct of assert terms
// as a single assert term
```

---

<sup>1</sup>Distributive Law

### 3.4 Formula Representation

Consider the formula  $f$  generated by ESBMC to be processed by the cache with  $t_0$  to  $t_{n-2}$  the assignments and  $t_{n-1}$  the disjunct of asserts as specified in Section 3.2 as well as the formula  $f'$  from Section 3.3.

$$f = t_0 \wedge t_1 \wedge \dots \wedge t_{n-1} \quad (3.31)$$

$$t_{n-1} = a_0 \vee a_1 \vee \dots \vee a_{m-1} \quad (3.32)$$

$$g = t_0 \wedge t_1 \wedge \dots \wedge t_{n-2} \quad (3.33)$$

$$f' = (g \wedge a_0) \vee (g \wedge a_1) \vee \dots \vee (g \wedge a_{m-1}) \quad (3.34)$$

Even though  $f$  and  $f'$  are polymorphic and will share the same satisfiability result if sent to an SMT solver, the time taken to compute the satisfiability might differ. Section 2.5.3 explains how the same formula presented in a different order can cause an increase or decrease in solving time while solving a polymorphic (among other) formula. To accurately assess the influence of the cache formulas sent to the SMT solver will be kept as consistent and as close to the original formula as possible. To achieve this the smallest alteration that can be made while taking advantage of results that show unsatisfiability of terms in the disjunct, is to only remove terms from the disjunct  $t_{n-1}$ . To do this still requires  $f'$  as the *asserts* found in the disjunct  $t_{n-1}$  could still undergo modifications need to be looked up independently from each other. Figure 3.2 shows a representation that would create a structure to be used as a solution.

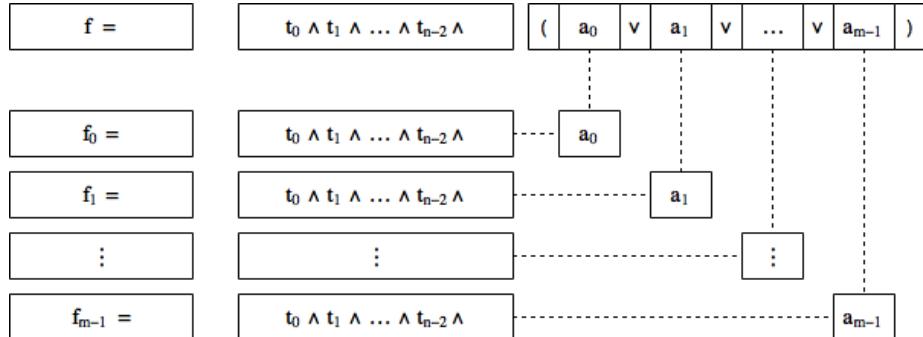


Figure 3.2: Formula Representation

In this representation, the formula  $f$  is broken up into the assignments and asserts as previously represented by  $g$  and  $t_{n-1}$ . The previous formula  $f'$  is now replaced with formulas  $f_0$  through  $f_{m-1}$  to represent  $L(t_{n-1})_0$  through  $L(t_{n-1})_{m-1}$  as follows.

$$f_i = g \wedge L(t_{n-1})_i \quad (3.35)$$

Each formula  $f_0$  through  $f_{m-1}$  can now go through formula modifications (Section 4) without changing the formula that will eventually be sent to the SMT solver,

resulting in the formulas  $f'_0$  through  $f'_{m-1}$  as shown in Figure 3.3. These formulas still represent the original terms that were found in  $t_{n-1}$ . In the following representation some of the formulas have had terms changed or removed by the formula modification step, with  $f'_0$  for example, having terms sliced out.

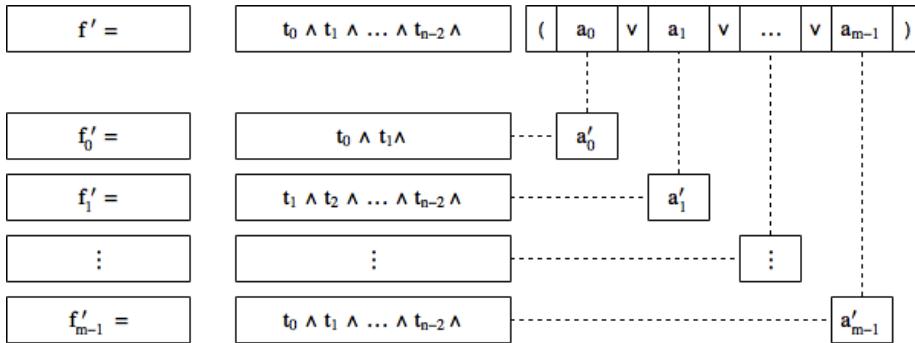


Figure 3.3: Formula representation with modification to lookup formulas

Finally, each formula  $f'_0$  through  $f'_{m-1}$  can be looked up (Section 3.6, 3.7 and 3.8) in the cache independently from each other, where a satisfiable lookup would result in the formula  $f$  being satisfiable, an unsatisfiable lookup would eliminate the corresponding clause, or in no lookup which would leave the original clause as is. The new formula to be sent to the SMT solver after the lookups would be represented as  $f'$ . Figure 3.4 illustrates a successful lookup where an unsatisfiable result was found and how it results in the formula  $f'$  after the corresponding clause in the disjunct from  $t_{n-1}$  was removed.

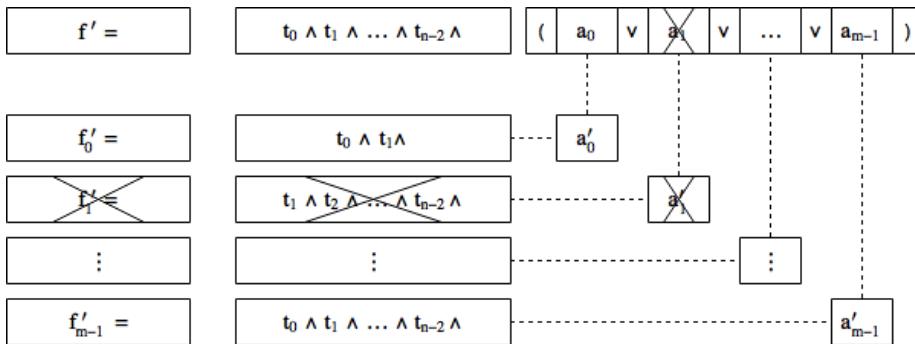


Figure 3.4: Formula Representation with successful unsatisfiable lookup

A suitable structure would need to be specified to reflect these different aspects and formulas to be handled in future operations. Starting with the structure we already have of  $f$ ,  $g$ ,  $t_{n-1}$  and  $f_i$ .

$$f = g \wedge t_{n-1} \quad (3.36)$$

$$t_{n-1} = a_0 \vee a_1 \vee \dots \vee a_{m-1} \quad (3.37)$$

$$g = t_0 \wedge t_1 \wedge \dots \wedge t_{n-2} \quad (3.38)$$

$$f_i = g \wedge L(t_{n-1})_i \quad (3.39)$$

Each formula  $f_i$  needs to be paired with its corresponding assert  $a_i$  statement so that the formula  $f_i$  may undergo modifications during the lookup process while maintaining  $a_i$  in its original form if it needs to be added to the disjunct of asserts to be sent to the SMT solver. A tuple  $d_i$  is created to pair them together as well as the operation  $A(d)$  to access the assert value in the tuple and  $F(d)$  to access the formula in the tuple.

$$d_i = (f_i, a_i) \quad (3.40)$$

$$A(d_i) = f_i \quad (3.41)$$

$$F(d_i) = a_i \quad (3.42)$$

A list of these tuples  $d_i$  need to be maintained along with the original conjunct of assignments to complete the structure. Let the list  $l$  denote the list of tuples  $d_i$  and  $r$  the tuple of the original conjunct of assignment  $g$  and the list  $l$  with the operation  $L(r)$  to access the list of the tuple and  $G(r)$  to access the conjunct of assignments  $g$ .

$$l = [d_0, d_1, \dots, d_{m-1}] \quad (3.43)$$

$$r = (g, l) \quad (3.44)$$

$$L(r) = l \quad (3.45)$$

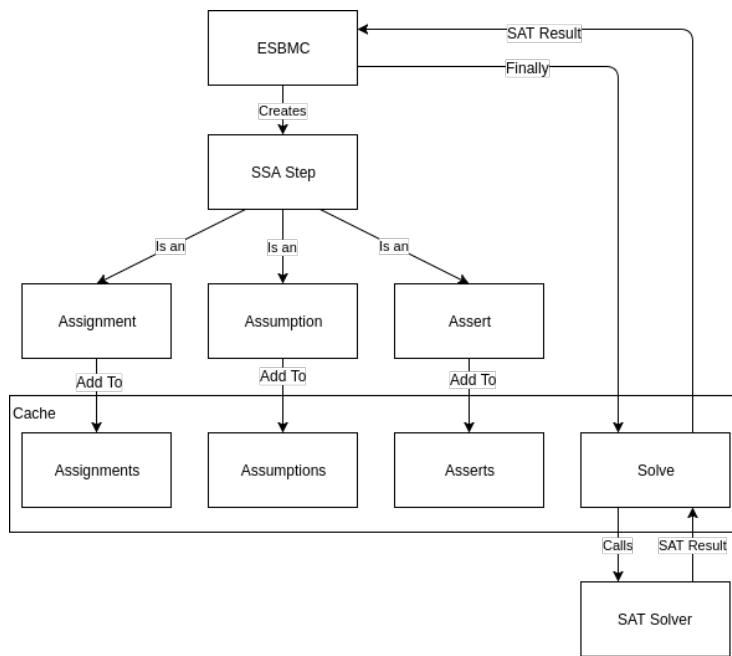
$$G(r) = g \quad (3.46)$$

A successful lookup for  $f_i$  with an unsatisfiable result would result in  $d_i$  being removed from  $l$ . Finally, a formula  $f'$  can be constructed to be sent to the SMT solver by taking  $g$  and the disjunct of all remaining  $a_i$  in the tuples in  $l$ .

$$f' = G(r) \wedge \left( \bigvee_{i=0}^{|L(r)|} A(L(r)_i) \right) \quad (3.47)$$

There might arise a situation where the formula being looked up wants to be used as the formula to be sent to the SMT solver. A program option is made available to override the default construction of the formula. Instead of the previously stated procedure, the formula  $f'$  to be sent to the SMT solver would then be constructed as follows:

$$f' = \bigvee_{i=0}^{|L(r)|} F(L(r)_i) \quad (3.48)$$



**Figure 3.5:** Point of interception of formulas before they are sent to the SMT solver

### ESBMC Program Options

```
--solve-lookup // Solve a disjunct of modified formulas
// that were used during cache lookup
```

## 3.5 Intercepting formulas between ESBMC and SMT Solvers

Section 2.5.3 shows how altering the formula sent to the SMT solver has an influence on the amount of time it takes for a satisfiability result to be reached. In order to give the most accurate assessment possible of the influence the cache will have, the only alterations allowed to be made to the formula is that of removing clauses from the disjunct of asserts. All the assignments generated by ESBMC still needs to be sent to the SMT solver, but the cache does use them in different ways to help with the lookup process. To accommodate this all assignments, assumptions and asserts are sent to the cache directly to be handled internally while introducing a call to solve the passed assignments, assumptions and asserts. The cache will then relay the information from the SMT solver back to ESBMC after it has taken the appropriate actions and decided what needs to be solved in the first place. Figure 3.5 shows the new flow from SSA Steps to the SMT solver where the cache will be used as an intermediary.

### 3.6 Storing Formulas For Exact Matches

A fundamental requirement of a cache is that it must be fast during both storing and retrieving of information (or depending on the expected ratio of which operation will be more frequent, only one might need to operate fast). For the purpose of this evaluation all results will be stored in memory inside of ESBMC, with the potential to use an external storage mechanism later to more effectively keep the results persistent across runs. A key value structure will be sufficient where the formula  $f$  provided will be stored along with its satisfiability. As previously stated, the only results that are of interest to store, are unsatisfiable results, as a satisfiable result would cause the end of the verification.

The following behavior is expected from the cache where the formula  $f$  is derived from representation in Section 3.4.

*Lookup / Store*

For the rest of this document, let the notation  $\text{exact-lookup}(f)$  represent the lookup of the exact formula  $f$  and  $\text{exact-store}(f, \{\text{unsat}\})$  the storing of the exact formula  $f$  with the specified satisfiability (limited to unsatisfiable currently).

$$f = t_0 \wedge t_1 \wedge \dots \wedge t_{n-1} \quad (3.49)$$

$$\text{exact-lookup}(f) \rightsquigarrow \text{unsat} \mid \text{none} \quad (3.50)$$

$$\text{exact-store}(f, \{\text{unsat}\}) \quad (3.51)$$

For the lookup to be fast, an appropriate key must be selected. The naive approach would use the entire formula as a key, but this will quickly become slow as the number of entries in the key value map increases as well as the increase in formula size. The data structure representing formulas in ESBMC provides access to a hash value, called the `crc` value for the formula. This value is represented by an unsigned integer, but is not a perfect hash (there is a possibility for a collision). Using the `crc` value will greatly decrease the storing and lookup time for a formula, but will not give us absolute certainty that the result that gets returned is the satisfiability of the formula being looked up.

A combination of the two approaches ensures good performance while ensuring accuracy. Two keys are used when looking up a formula, where the `crc` value returns a separate key value object where the next key is the entire formula again. There is scope to use other properties of the formulas to introduce additional steps to avoid using the entire formula as much as possible, for example, the number of terms in a formula. These additional possibilities are not explored further in this evaluation but Section 5.4.11 shows that there is time to be made up per iteration with regards to the amount of overhead in the current implementation of the cache.

## Program Options

```
--lookup          // Activates full formula matching
--full-lookup    // Uses full formula as key
--crc-lookup     // Uses only crc value as key
--crc-full-lookup // Uses crc value as first key and
                  // full formula as secondary key
                  // Default setting
```

## 3.7 Storing Formulas For Subset Matches

Consider the following formula:

$$f = a \wedge b \wedge \neg a$$

The formula  $f$  is unsatisfiable due to the terms  $a \wedge \neg a$ . The term  $b$  has no influence on the satisfiability of the formula, and adding any other conjunctive terms will still give the same unsatisfiable result. This gives way to an approach to better reuse unsatisfiable formulas that are already in the cache. If the formula  $f$  is being looked up in the cache, and there exists a formula  $g$  that is unsatisfiable, with  $g$  being a subset of the formula  $f$ , then the formula  $f$  is unsatisfiable. The reverse, however, is not true, as removing clauses from the conjunct could change the satisfiability of the formula.

$$\text{exact-lookup}(g) \rightsquigarrow \text{unsat} \implies \text{exact-lookup}(f) \rightsquigarrow \text{unsat} \mid g \subseteq f$$

### Lookup / Store

For the rest of this document, let the notation  $\text{subset-lookup}(f)$  represent the lookup of a formula  $f$  by means of subset matching and  $\text{subset-store}(f, \{\text{unsat}\})$  the storing of the formula  $f$  with the specified satisfiability (limited to unsatisfiable currently) for future subset matching.

### 3.7.1 Proving a Subset Match

To find out if the satisfiability of formula  $f$  is already known, a comparison must be done to a formula  $g$  that is unsatisfiable to see if  $g$  is a subset of  $f$ . Consider the following conjunctive formulas  $f$  and  $g$  with  $T$  the set of all terms contained in  $f$  and  $U$  the set of all terms contained in  $g$ .

$$f = t_0 \wedge t_1 \wedge \dots \wedge t_{n-1}, \quad T = \{t_0, t_1, \dots, t_{n-1}\} \quad (3.52)$$

$$g = u_0 \wedge u_1 \wedge \dots \wedge u_{m-1}, \quad U = \{u_0, u_1, \dots, u_{m-1}\} \quad (3.53)$$

To show that  $g$  is a subset of  $f$  the following needs to be provided:

- A conjunctive formula  $f'$  containing only terms from  $f$ . Let  $T = \{t_0, t_1, \dots, t_{n-1}\}$  and  $T' = \{t'_0, t'_1, \dots, t'_{m-1}\}$ , with  $T' \subseteq T$
- An injective function  $p(t) \mapsto d$  that maps each term in  $f'$  to a term in  $g$ .
- An injective function  $q(v) \mapsto v'$  that contains a renaming for each unique variable found in  $f'$  to a variable found in  $g$ .

- A function  $r(t, q) \mapsto t'$  that takes a term  $t$  and renames every variable in the term according to the injective function  $q$ .

For each term  $t'_i$  in  $f'$ , if it follows that

$$r(t'_i, q) = p(t'_i) \mid \forall t'_i \in T'$$

then the information provided was sufficient to prove that  $g \subseteq f$  by producing a mapping for each term as well as a renaming function that will create  $f'$ .

With the introduction of injective functions, the following notation will be used to access specific values of an injective function  $q$ :

$$D(q) = \text{Domain of } q$$

$$R(q) = \text{Range of } q$$

Since the cache is storing multiple formulas, it is unlikely that there will be only one formula  $g$  to compare  $f$  to. Instead there will be a set  $G = \{g_0, g_1, \dots, g_{o-1}\}$  to compare to, containing all the unsatisfiable formulas stored so far. To find a cache hit then becomes:

$$\text{exact-lookup}(g) \rightsquigarrow \text{unsat} \implies \text{exact-lookup}(f) \rightsquigarrow \text{unsat} \mid g \subseteq f, g \in G$$

The set  $G$  of unsatisfiable formulas grows quite rapidly during each subsequent unwinding. If there exists no formula  $g$  that is a subset of formula  $f$  then each formula will be put through the comparison process which can be quite time-consuming. Restricting the size of  $G$  becomes an integral part to the cache efficiency.

### 3.7.2 Decreasing The Size Of $G$

Before evaluating all unsatisfiable formulas from  $G$  as potential subset matches for formula  $f$ ,  $G$  should be stripped of formulas that are unlikely to be a subset of  $f$ . To do this, a term is identified that is in  $f$  but only in some of the formulas in  $G$ , making it easy to exclude those formulas that do not contain it. An assumption is made here that the term being identified is crucial to the unsatisfiability of the formula, otherwise potentially matching formulas in  $G$  are excluded. This term can then be used as the key in the cache, splitting  $G$  up into smaller sets of formulas where they key ensures the presence of the term being looked up. Let  $G_t$  denote the subset of  $G$  for which each formula  $g \in G$  contains the term  $t$ .

$$G = G_{t_0} \cup G_{t_1} \cup \dots \cup G_{t_{n-1}} \tag{3.54}$$

$$\text{Glookup}(t_n) = G_{t_n} \tag{3.55}$$

A good candidate for this clause, is the assert statement that is present as the last term of the formula. The assert is considered a good candidate as it contains the property violation which the program is attempting to reach. If this state was previously unreachable it would be paired with a list of assignments to that state. If it was previously shown to be unreachable and the current list of assignments have more terms in the conjunct then it will still be unreachable. By renaming just the assert, any other

assert with the same structure but different variables names will be identified and a list of these formulas will be provided to be compared against. This has an added benefit where the variable map  $q$  (Section 3.7.1 requirement as proof) can be populated initially with the variables from the last term in  $f$  and the last term in  $g$  (the assert term) creating a good starting point to assist the Rename Filter (Section 3.7.4).

$$f = t_0 \wedge t_1 \wedge \dots \wedge t_{n-1} \quad (3.56)$$

$$\text{Glookup}(t_{n-1}) = G_{t_{n-1}} \quad (3.57)$$

$$\text{exact-lookup}(g) \rightsquigarrow \text{unsat} \implies \text{exact-lookup}(f) \rightsquigarrow \text{unsat} \mid g \subseteq f, g \in G_{t_{n-1}} \quad (3.58)$$

### 3.7.3 Finding A Subset Match

To determine if formula  $g$  is a subset of formula  $f$ , an efficient way to match their respective terms is required. Let  $f$  and  $g$  be represented as follows:

$$f = t_0 \wedge t_1 \wedge \dots \wedge t_{n-1} \quad (3.59)$$

$$g = u_0 \wedge u_1 \wedge \dots \wedge u_{m-1} \quad (3.60)$$

A boolean table can be created to show the potential matches form the clauses in  $g$  to the clauses in  $f$ , with a value of 1 indicating a potential match and a 0 not a potential match. At the start all clauses can be mapped to each other, a series of Filters (Section 3.7.4) will then start eliminating potential matches if they have failed the given filter's respective criteria.

Initial					After Some Filters					
	$t_0$	$t_1$	$\dots$	$t_{n-1}$		$u_0$	$t_0$	$t_1$	$\dots$	$t_{n-1}$
$u_0$	1	1	$\dots$	1		$u_0$	0	0	$\dots$	1
$u_1$	1	1	$\dots$	1		$u_1$	1	0	$\dots$	1
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$u_{m-1}$	1	1	$\dots$	1		$u_{m-1}$	1	1	$\dots$	0

An example match where  $g \subseteq f$

	$t_0$	$t_1$	$t_2$	$t_3$	$t_4$
$u_0$	1	0	0	0	0
$u_1$	0	0	0	1	0
$u_2$	0	0	0	0	1

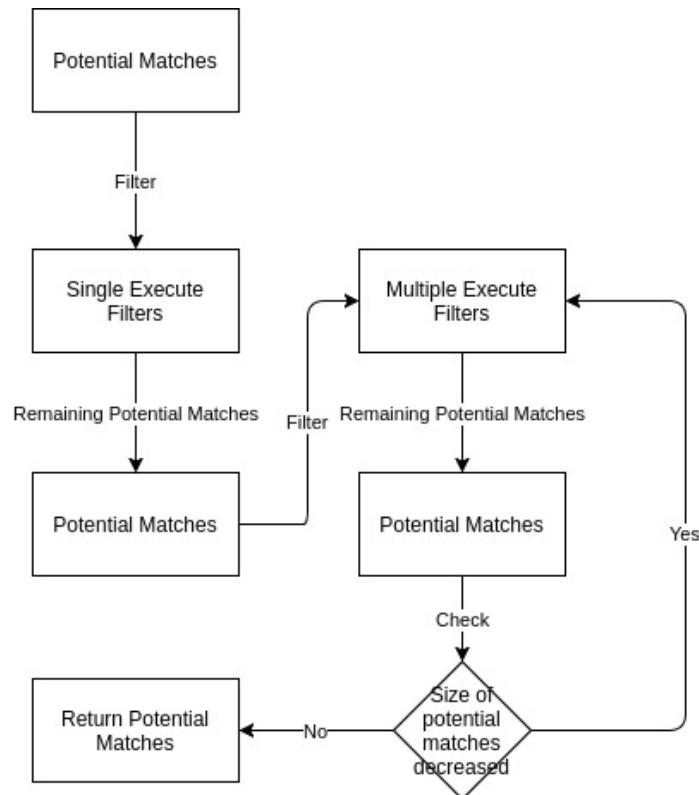
Mapping:  $u_0 \rightarrow t_0, u_1 \rightarrow t_3, u_2 \rightarrow t_4$

For a successful match, each term in  $g$  must be mapped to exactly one term in  $f$  (no two terms can map to same term in  $f$  and no term can be mapped to more than once). If such a mapping can be found it can be used as the injective function  $p$  required by Section 3.7.1.

### 3.7.4 Filters

To assist in the elimination of potential matches, multiple filters are created that can reason about the properties in two terms to determine if they are a potential match or if they are definitely not a potential match. These filters are categorized into two separate groups namely single execution and multiple execution. Single execution filters need only be applied once as the criteria to eliminate matches do not change based on the other available potential matches. Multiple execution filters are expected to be run multiple times as the filter can exclude more potential matches if the set of potential matches has changed since the last time the filter has been executed.

Figure 3.6 demonstrates the execution cycles of the filters. For the filters that execute multiple times, a record is kept to see if there has been any changes since its last execute and the matching algorithm will only stop once an iteration has completed where no filter was able to modify the set of potential matches for all terms.



**Figure 3.6:** Subset Matching process with regards to Filter operations

### Rename Filter

The first filter to be introduced is the renaming filter. This filter forms a fundamental part of the subset matching as it provides the injective function  $q$  containing the variable renaming as is required by Section 3.7.1 to prove a subset match. The filter takes two approaches to help finding and eliminating matches.

The first approach takes every term  $u$  in  $g$  and compares it to every term  $t$  in  $f$ , if it has not previously been shown that these terms can't match (0 in the table layout). If term  $t$  and term  $u$  both contain a value  $v$  that is a variable, a lookup can be done in the map  $q$  to see if the expected value is found. If the function  $eval_0$  evaluates to *False*, then a contradiction has been found and the terms  $t$  and  $u$  cannot be a potential match. The function first checks if neither the variable from  $t$  is in the domain and the value from  $u$  is not in the range, this indicates that these variables do not currently form part of a mapping and can still be mapped to each other. If both have appeared somewhere in a mapping, then they must map to each other otherwise more than one mapping exists and a contradiction has occurred.

$$eval_0(t, u, q) = \begin{cases} \text{True}, & V(t) \notin D(q) \wedge V(u) \notin R(q) \\ q(v_t) = V(u), & V(t) \in D(q) \wedge V(u) \in R(q) \\ \text{False} & V(t) \in D(q) \vee V(u) \in R(q) \end{cases} \quad (3.61)$$

Otherwise if term  $t$  and term  $u$  do not contain variables but both contain the same amount of children in their list  $L(t)$  and  $L(u)$  (Shape Filter will force possible matches to share an operand  $o$  and same amount of children), then for each child term at position  $i$  in each each list, repeat the process with the new term  $t'$  and  $u'$  with any contradiction in subsequent comparisons causing the original terms  $t$  and  $u$  to not be potential matches. The functions  $eval_1$  and  $eval$  illustrate these extra steps with  $eval$  representing the entry point for the original comparison of terms  $t$  and  $u$  with the variable map  $q$  containing all the current known mappings.

$$eval_1(t, u, q) = \bigwedge_{i=0}^{|L(t)|} eval(L(t)_i, L(u)_i, q) \quad (3.62)$$

$$eval(t, u, q) = \begin{cases} eval_0(t, u, q), & V(t) \neq \text{null} \\ eval_1(t, u, q), & V(t) = \text{null} \end{cases} \quad (3.63)$$

The second approach, counts the amount of potential matches that term  $u$  has. If there is only 1 potential match, then those terms must match, otherwise  $g$  is not a subset of  $f$ . Since there is only 1 potential match, all variables in term  $u$  is mapped to term  $t$  in the variable map. If at any time there already exists a different mapping in the variable map, compared to what is currently being matched for terms  $t$  and  $u$ , then there is a contradictory mapping and formula  $g$  cannot be a subset of  $f$ .

This filter needs to be run multiple times as  $q$  can grow and influence the results from the first approach and during other comparisons potential matches can be eliminated, potentially ending up with a single match that will then trigger approach 2. Listing 3.1 and Listing 3.2 provide the pseudo code for these approaches.

```

1 def match(term1, potential_matches, variable_map):
2     for term2 in potential_matches:
3         if (!match(term1, term2, variable_map)):
4             potential_matches.remove(term2)
5
6 def match(term1, term2, variable_map):
7     if (is_variable(term1) and is_variable(term2)):
8         if variable_map[term1] == term2:
9             return True
10        else:
11            return False
12    else:
13        for child1, child2 in term1.children, term2.children:
14            if (!match(child1, child2, variable_map)):
15                return False
16
17 return True

```

**Listing 3.1:** Pseudo Code for approach 1 of Range Filter

```

1 def map(term1, potential_matches, variable_map):
2     if (potential_matches.size() == 1):
3         term2 = potential_matches[0]
4     if (!map(term1, term2)):
5         potential_matches.remove(term2)
6
7 def map(term1, term2, variable_map):
8     if (is_variable(term1) and is_variable(term2)):
9         // See if there is already a mapping
10        if (variable_map.contains(term1)):
11            // Make sure the variable map stays consistent
12            if (variable_map[term1] != term2):
13                return False
14        else:
15            variable_map[term1] = term2

```

**Listing 3.2:** Pseudo Code for approach 2 of Range Filter

### Shape Filter

A simple method of eliminating potential matches that clearly cannot match, is to eliminate potential matches between any terms that do not have the same type (operation or in case of variable, data type) or that do not share the same amount of children at any point. This filter only needs to be run once since it will eliminate all the potential matches in one execution, subsequent executions will not encounter terms that have changed in shape.

```

1 def match(term1, potential_matches):
2     for term2 in potential_matches:
3         if (!match(term1, term2)):
4             potential_matches.remove(term2)
5
6 def match(term1, term2, variable_map):
7     if (type(term1) != type(term2)):
8         return False
9     if (len(term1.children) != len(term2.children)):
10        return False
11    for child1, child2 in term1.children, term2.children:
12        if (!match(child1, child2, variable_map)):

```

```

13     return False
14     return True

```

**Listing 3.3:** Pseudo Code for Shape Filter

### Exploiting Domain Knowledge

During the unwinding process of bounded model checking an underlying transition system is created to represent the system for the unwind bound  $k$ . A transition from state  $s_i \rightarrow s_j$  for bound  $k$ , cannot appear earlier along a path compared to the same path for bound  $k - 1$ . The state transition will either appear at the same position along the path, or it will appear later on the path (or a branch of the path). This allows for some assumptions to be made in subsequent filters where there is some consistency in the order in which terms are produced by ESBMC.

### Range Filter

If it can be assumed that terms do not appear earlier in bound  $k$  than they did in bound  $k - 1$  then this information can be used to create a filter that eliminates the potential matches that appear earlier. Furthermore, since all terms need to have a match, a range is created of terms that can potentially match to ensure there are still potential matches for subsequent terms. Consider the formulas  $f$  and  $g$  where it will be attempted to prove that  $g \subseteq f$

$$f = t_0 \wedge t_1 \wedge \dots \wedge t_{n-1} \quad (3.64)$$

$$g = u_0 \wedge u_1 \wedge \dots \wedge u_{m-1} \quad (3.65)$$

If the size of  $f$  is  $n$  and the size of  $g$  is  $m$  with  $n \geq m$ , then the range  $r$  of terms that is available for any single term in  $g$  as a potential match is  $r = n - m + 1$ . For any term  $u_i$  in  $g$  the potential matches can be limited to  $\{t_i, t_{i+1}, \dots, t_{i+r}\}$  with all other potential matches eliminated. The pseudo code for the shape filter can be found in Listing 3.4.

```

1 Pseudo Code:
2
3 // m = len(f)
4 // n = len(g)
5 def match(m, n, term1, potential_matches):
6     range = m - n
7     start = pos(term1)
8     end = start + range
9     for clause2 in potential_matches:
10         if !(pos(term2) >= start and pos(term2) <= end)):
11             potential_matches.remove(term2)

```

**Listing 3.4:** Pseudo Code for Range Filter

### Space enhancement for potential matches

If the range filter is used during the process of subset matching, it quickly eliminates a lot of potential matches in a very predictable way.

The original amount of potential matches for a single row was  $n$ , the amount of space that can now be saved for that row is  $n - r = n - (n - m + 1) = m - 1$ . This amount can be saved for every row giving a total of  $(m - 1)m$  which can be expressed as a fraction of the original amount of space required.

$$\frac{(m - 1)m}{nm} = \frac{m - 1}{n} \quad (3.66)$$

A new representation can now be created where for each row  $i$  the column  $j$  reflects the match between the two terms  $t_i$  and  $t_{i+j}$ . This representation can be visualized as follows:

	$t_0$	$t_1$	$t_2$	$t_3$	$t_4$	$t_5$		$u_0$	$t_0$	$t_1$	$t_2$
$u_0$	1	1	1	0	0	0	→	$u_1$	$t_1$	$t_2$	$t_3$
$u_1$	0	1	1	1	0	0		$u_2$	$t_2$	$t_3$	$t_4$
$u_2$	0	0	1	1	1	0		$u_3$	$t_3$	$t_4$	$t_5$
$u_3$	0	0	0	1	1	1					

### Forwards And Backwards Filter

	$t_0$	$t_1$	$t_3$	$t_4$
$u_1$	0	1	1	1
$u_2$	0	1	1	1
$u_3$	0	1	1	1

Let us take this as our current state of matching. Although it may currently seem like the 3 formulas can be mapped to any of the other 3 formulas, the answer is actually already quite evident as:

	$t_0$	$t_1$	$t_3$	$t_4$
$u_1$	0	1	0	0
$u_2$	0	0	1	0
$u_3$	0	0	0	1

We can derive this with the following statement. If the first potential match for  $u_1$  is  $t_1$ , and we assume it to be the correct match, then the first potential match for  $u_2$  must be at least the first match of  $t_1 + 1$ , thus the first match for  $u_2$  is  $t_2$ . This process can be applied since we are assuming that terms are added to the conjunct in a specific order. During subsequent unwindings this order must be maintained for the terms that were encountered during the previous unwinding. This is forward matching and results in the following table:

	$t_0$	$t_1$	$t_3$	$t_4$
$u_1$	0	1	1	1
$u_2$	0	0	1	1
$u_3$	0	0	0	1

But the same logic can be used to imply matches backwards, otherwise there will be nothing left to match.

	$t_0$	$t_1$	$t_3$	$t_4$
$u_1$	0	1	0	0
$u_2$	0	0	1	0
$u_3$	0	0	0	1

### Final Note

All these filters combined gives us a method to find potential matches. It does not guarantee a match, since there might be a case with no violations, but without a 1 to 1 mapping, from here more filters and techniques would need to be introduced. The filters never guess a return value, either a complete mapping is provided or no mapping at all. When no mapping is returned it does not necessarily mean no mapping exists, just that the filters could not determine one.

### 3.7.5 ESBMC Program Options

```
--subset-lookup          // Activates subset formula matching
--full-subset-lookup    // Uses full formula as key
--crc-subset-lookup     // Uses only crc value as key
--crc-full-subset-lookup // Uses crc value as first key and
                        // full formula as secondary key
--use-rename-filter
--use-shape-filter
--use-range-filter
--use-backwards-and-forwards-filter
```

## 3.8 Unsatisfiable Cores

### *Lookup / Store*

For the rest of this document, let the notation  $\text{unsat-core-lookup}(f)$  represent the lookup of a formula  $f$  by means of subset matching and  $\text{unsat-core-store}(f, \{\text{unsat}\})$  the storing of the formula  $f$  with the specified satisfiability (limited to unsatisfiable currently) by means of subset matching. It is expected that all formulas used during these operations are marked as the unsat core from the relevant SMT solver.

When the SMT solver returns an unsatisfiable result for a formula  $f$ , it is often the case that not all the terms in the formula were needed to find the unsatisfiable result. Section 3.7 makes use of a similar principle where a subset of a formula is already known to be unsatisfiable. This conjunct of terms that caused an unsatisfiable result to be obtained by the SMT solver is called the *unsat-core*. A minimal unsat-core (which is not necessarily supplied by the SMT solver) would constitute a conjunct of terms where the removal of any term would result in a satisfiable formula. A formula could potentially have many different *unsat-cores* with the term *minimal unsat-core* referring to the *unsat-core* with the least amount of clauses in it. If the *unsat-core* or *minimal unsat-core* of the formula  $f$  can be stored instead of the formula  $f$ , an increase in

hits can potentially be expected during subset lookups (Section 3.7).

Looking at the code listing from Section 2.3 again and the subsequent SMT formula for bounds  $k = 1$ . Listing 2.2 from Section 2.3 is considered again with the following SMT formulas created:

SAT equation for  $k = 1$

```
(= x 1)
(= y 2)
(= z 0)
(= guard1 (< z 1))
(= x1 (+ x 1))
(= z1 (+ z 1))

(and guard1 not((!= x y)))
```

The SMT solver is attempting to find satisfiable variable assignments for either of the following formulas (after variable propagation as implemented by Z3):

Either  $f_0$ :

```
(= guard1 True)
(= guard2 False)
(and guard1 not((!= 1 2)))(and guard2 not((!= 2 2))))
```

Or  $f_1$ :

For  $f_0$  the term  $\text{not}((!= 1 2))$  evaluates to *False* and subsequently the term  $(\text{and guard2 not}((!= 2 2)))$  also evaluates to *False*. For  $f_1$  the term  $(\text{and guard2 not}((!= 2 2)))$  evaluates to *False* ( $= \text{guard2 False}$ ). From the example it can already be seen that separate parts of the formula caused the unsat-cores. The following listing will identify the original clauses that form part of each unsat-core.

Either  $f_0$ :

```
(= x 1) // unsat
(= y 2) // unsat
(= z 0) // unsat
(= guard1 (< z 1)) // unsat
(= x1 (+ x 1))
(= z1 (+ z 1))
(= guard2 (< z1 1))
(= x2 (+ x1 1))
(= z2 (+ z1 1))
(and guard1 not((!= x y)))(and guard2 not((!= x1 y))))
// unsat
```

Or  $f_1$ :

```
(= x 1) // unsat
(= y 2) // unsat
(= z 0) // unsat
(= guard1 (< z 1)) // unsat
(= x1 (+ x 1)) // unsat
(= z1 (+ z 1)) // unsat
(= guard2 (< z1 1)) // unsat
(= x2 (+ x1 1))
(= z2 (+ z1 1))
// unsat
```

From both formulas it can be seen that the terms related to the unsat-core is smaller than the original formulas. Formula modifications like Variable Dependent Formulas (Section 4.2) would not have been able to slice the formulas down to the

size of the unsat-cores.

The SMT solver however would have returned the unsat-core from the original formula, namely:

```
(= x 1) // unsat
(= y 2) // unsat
(= z 0) // unsat
(= guard1 (< z 1)) // unsat
(= x1 (+ x 1)) // unsat
(= z1 (+ z 1)) // unsat
(= guard2 (< z1 1)) // unsat
(= x2 (+ x1 1))
(= z2 (+ z1 1))

(or // unsat
  (and guard1 not((!= x y)))
  (and guard2 not((!= x1 y))))
```

Some of the information regarding the individual unsat-cores has now been lost for the formula  $f_0$ . Since all statements within the disjunct were unsatisfiable, all clauses within the disjunct form part of the unsat-core. To find the unsat-core of each part of the disjunct,  $f_0$  and  $f_1$  would have to be sent to the solver separately.

To understand how Z3 returns an unsat-core the previous example will be used again. Z3 returns a list of labels that formed the unsat-core, with each label referring to a clause in the formula. Once labeled for Z3 the SMT formula would change to the following

```
(=> label0 (= x 1))
(=> label1 (= y 2))
(=> label2 (= z 0))
(=> label3 (= guard1 (< z 1)))
(=> label4 (= x1 (+ x 1)))
(=> label5 (= z1 (+ z 1)))
(=> label6 (= guard2 (< z1 1)))
(=> label7 (= x2 (+ x1 1)))
(=> label8 (= z2 (+ z1 1)))

(=> label9 (or
  (and guard1 not((!= x y)))
  (and guard2 not((!= x1 y)))))
```

It can already be seen that the formula grew in size, and as can be seen in Section 5.4.9, the formula does take longer to solve compared to the original formula without the labels. A list of labels that formed part of the unsat core is then returned if the satisfiability of the formula was false.

### Storing the unsat core

There now exists the possibility where we do not have a key to use in the cache. If the unsat-core does not contain the assert, then we do not have a place to map it to. Also, as mentioned initially in the subset lookup description, it would not be efficient to keep a list of all unsat cores as the list will grow linearly and solving time quadratically as each term from  $f$  would have to be checked for a potential match of each term in each formula that is stored as unsatisfiable. Since there is no extra information about the terms being returned there is no obvious solution to this problem to be implemented for this version of the cache.

An initial best attempt approach could be as follows: When the unsat core is returned, check if the assert label was part of the returned values, if it was, the new formula can be constructed and still mapped using the assert as the key. If the assert did not form part, then we just map it to a constant key, in this case, just True. A side effect this will also have is removing the original population of the renaming map during the matching process for cases where we have no assert present.

### ESBMC Program Options

```
--unsat-core-lookup          // Activates subset formula matching
--full-unsat-core-lookup    // Uses full formula as key
--crc-unsat-core-lookup     // Uses only crc value as key
--crc-full-unsat-core-lookup // Uses crc value as first key and
                           // full formula as secondary key
```

## 3.9 Store formulas before results are known

If the cache is primarily used for reuse within a single run, then the only results worth storing are unsatisfiable results. If a satisfiable assignment is found, the program will terminate with a counterexample and the results cleared from the cache. This allows for formulas to be stored as soon as the lookup fails to find it in the cache.

Consider the formula  $f$  to be sent to the SMT solver with the formulas  $f_0$  and  $f_1$  derived from  $f$  to be looked up in the cache to see if any part of the disjunct can be removed.

$$f = (a < 0) \wedge (b < 0) \wedge ((a > 10) \vee (b > 10)) \quad (3.67)$$

$$f_0 = (a < 0) \wedge (a > 10) \quad (3.68)$$

$$f_1 = (b < 0) \wedge (b > 10) \quad (3.69)$$

$$f_0 \neq f_1 \quad (3.70)$$

Let the cache be empty when the process starts, neither  $f_0$  nor  $f_1$  would have been found in the cache, causing  $f$  to be sent to the SMT solver as is.

$$\text{lookup}(f_0) \rightsquigarrow \text{none} \quad (3.71)$$

$$\text{lookup}(f_1) \rightsquigarrow \text{none} \quad (3.72)$$

$$\text{store}(f_0, \text{unsat}) \quad (3.73)$$

$$\text{store}(f_1, \text{unsat}) \quad (3.74)$$

Now consider the scenario where the formulas are processed by the *Variable Renaming* step (Section 4.2) resulting in  $f'_0$  and  $f'_1$ , then the formulas  $f'_0$  and  $f'_1$  would have been equal.

$$f'_0 = (v_0 < 0) \wedge (v_0 > 10) \quad (3.75)$$

$$f'_1 = (v_0 < 0) \wedge (v_0 > 10) \quad (3.76)$$

$$f'_0 = f'_1 \quad (3.77)$$

It is clear that only one of these need to be solved as they will share the same satisfiability result. Consider the cache is empty again when the experiment starts, neither  $f'_0$  nor  $f'_1$  would have been found resulting in  $f$  still being sent to the SMT solver as is.

$$\text{lookup}(f'_0) \rightsquigarrow \text{none} \quad (3.78)$$

$$\text{lookup}(f'_1) \rightsquigarrow \text{none} \quad (3.79)$$

$$\text{store}(f'_0, \text{unsat}) \quad (3.80)$$

$$\text{store}(f'_1, \text{unsat}) \quad (3.81)$$

Enabling results to be stored before they are known by enabling this formula modification, will store a formula as unsatisfiable as soon as the lookup failed. For the previous example,  $f'_0$  would not have been found in the cache, resulting in  $f'_0$  to be stored as *unsat*, when  $f'_1$  gets looked up afterwards, a cache hit will be registered before the result of  $f'_0$  is confirmed by the SMT solver.

$$\text{lookup}(f'_0) \rightsquigarrow \text{none} \quad (3.82)$$

$$\text{store}(f'_0, \text{unsat}) \quad (3.83)$$

$$\text{lookup}(f'_1) \rightsquigarrow \text{unsat} \quad (3.84)$$

$$f' = (a < 0) \wedge (b < 0) \wedge ((a > 10)) \quad (3.85)$$

Even though  $f'_0$  and  $f'_1$  could have easily been compared without storing the result of  $f'_0$  in the cache and then looking up  $f'_1$ , the technique accommodates more cache hits during subset lookups that would have failed just comparing  $f'_0$  to  $f'_1$ .

Consider the slightly modified formula  $f$  where the term ( $b \neq 2$ ) was added.

$$f = (a < 0) \wedge (b < 0) \wedge (b \neq 2) \wedge ((a > 10) \vee (b > 10)) \quad (3.86)$$

$$f_0 = (a < 0) \wedge (a > 10) \quad (3.87)$$

$$f_1 = (b < 0) \wedge (b \neq 2) \wedge (b > 10) \quad (3.88)$$

$$f_0 \neq f_1 \quad (3.89)$$

After renaming the variables:

$$f'_0 = (v_0 < 0) \wedge (v_0 > 10) \quad (3.90)$$

$$f'_1 = (v_0 < 0) \wedge (v_0 \neq 2) \wedge (v_0 > 10) \quad (3.91)$$

$$f'_0 \neq f'_1 \quad (3.92)$$

$$\text{subset-lookup}(f'_0) \rightsquigarrow \text{none} \quad (3.93)$$

$$\text{subset-lookup}(f'_1) \rightsquigarrow \text{none} \quad (3.94)$$

$$\text{subset-store}(f'_0, \text{unsat}) \quad (3.95)$$

$$\text{subset-store}(f'_1, \text{unsat}) \quad (3.96)$$

It is now no longer the case that  $f'_0$  and  $f'_1$  can be compared without the use of the cache, even though they are not equal, they will share the same satisfiability result if  $f'_0$  is shown to be unsatisfiable. By assuming  $f'_0$  will be unsatisfiable,  $f'_1$  can also be shown to be unsatisfiable by showing  $f'_0 \subseteq f'_1$  through subset matching (Section 3.7). By storing  $f'_0$  right after the lookup fails will allow the subset lookup operation to find a cache hit for  $f'_1$ .

$$\text{subset-lookup}(f'_0) \rightsquigarrow \text{none} \quad (3.97)$$

$$\text{subset-store}(f'_0, \text{unsat}) \quad (3.98)$$

$$\text{subset-lookup}(f'_1) \rightsquigarrow \text{unsat} \quad (3.99)$$

$$f' = (a < 0) \wedge (b < 0) \wedge (b \neq 2) \wedge ((a > 10)) \quad (3.100)$$

## ESBMC Program Options

```
--store-during-lookup
--store-during-subset-lookup
--store-during-unsat-core-lookup
```

## 3.10 Storing Incorrect Results

The strategy for caching and reusing results so far has revolved around the idea that the disjunct of asserts at the end of the formula sent to the SMT solver can be stored separately. This is based on the assumption that an unsatisfiable result from the SMT solver implied that each part of the disjunct was checked and no satisfiable assignment found, and thus all parts of the disjunct can be stored as unsatisfiable formulas in the cache.

Consider the following SMT formula:

```
(= a 0)
(= a 1)
(= guard1 (< a b))
(= guard2 (< c d))
(or
  guard1
  guard2)
```

The following formulas would have been created, checked for in the cache and eventually stored if they do not appear in the cache and the SMT solver returned an unsatisfiable assignment.

$$(a = 0) \wedge (a = 1) \wedge (a < b) \quad (3.101)$$

$$(c < d) \quad (3.102)$$

The first formula is unsatisfiable while the second one is satisfiable. The SAT solver will return a result of unsatisfiable though, causing both formulas to be stored as unsatisfiable in the cache. An option is made available in ESBMC that allows for the formula to be split into two separate solver calls, the first will evaluate all assignments first while the second will add on to the solver (the solver can still reuse some of its results).

### ESBMC Program Options

```
--solve-assignments-first // Call solver twice, checking
// satisfiability of assignments
// first
```



## Chapter 4

# Formula Modifications For Increased Reuse Potential

This chapter will explore multiple strategies to alter the formulas that are being looked up to increase the chances of finding a cache hit. Operations will be performed on each formula  $f_i$  in  $d_i$  as described by the structure in Section 3.4.

$$f = g \wedge t_{n-1} \quad (4.1)$$

$$t_{n-1} = a_0 \vee a_1 \vee \dots \vee a_{m-1} \quad (4.2)$$

$$g = t_0 \wedge t_1 \wedge \dots \wedge t_{n-2} \quad (4.3)$$

$$f_i = g \wedge L(t_{n-1})_i \quad (4.4)$$

$$d_i = (f_i, a_i) \quad (4.5)$$

### 4.1 Expressions Relevant To Assert Only

During the formula construction, assignments, assumptions and asserts are encountered in lexicographical order based on how they were generated from the transition system. It can safely be assumed that as soon as an `assert` is encountered, only assignments and assumptions encountered up to that point have an influence on that assert.

Currently each formula  $f_i$  is constructed by creating a conjunct of the assignments  $g$  and the assert  $a_i$ . The assert  $a_i$  however only uses the assumptions that have been seen up to that point. The same idea can be done for the assignments encountered when the assert is encountered. If  $g_i$  denotes all the assignments that have been encountered when an assert  $a_i$  is encountered, then the formula  $f_i$  to be added with  $a_i$  into the tuple  $d_i$  can be constructed as follows:

$$g_i = t_0 \wedge t_1 \wedge \dots \wedge t_j \quad (4.6)$$

$$g = t_0 \wedge t_1 \wedge \dots \wedge t_j \wedge \dots \wedge t_{n-1} \quad (4.7)$$

$$f_i = g_i \wedge a_i \quad (4.8)$$

$$d_i = (f_i, a_i) \quad (4.9)$$

## 42 CHAPTER 4. FORMULA MODIFICATIONS FOR INCREASED REUSE POTENTIAL

This will eliminate a lot of terms that are not needed to determine the satisfiability of  $a_i$  and will prove to be a vital part of getting a good amount of reuse during the evaluation in Section 5.

### ESBMC Program Options

```
--lookup-when-assert // Builds the formula to be looked up
                     // with the assignments that are currently
                     // available
```

## 4.2 Renaming Variables within Formulas

When storing and retrieving formulas from a formula cache, it is extremely unlikely that the formulas will have a consistent naming convention with regards to variable names that will allow for easy matching. To ensure that all formulas that are polymorphic to each other with respects to variable names are identified in the cache, an option is provided to rename all variables within a formula  $f$  in a standardized manner to produce a new formula  $f'$ . The operation  $\text{rename}(t, q)$  will take a term  $t$  and injective map  $q$  that will be used to lookup variables and replace them with the appropriate renamed variable. If no lookup exists within  $q$  then a new variable will be created. Variables are assigned new values in the order in which they were found through a depth-first exploration. Listing 4.1 provides the pseudo code for this operation.

```

1 def rename(t, q):
2     counter = 0
3     rename(t, q, counter)
4
5 def rename(t, q, counter):
6     if is_value(t):
7         return value
8     elif is_variable(t):
9         if t not in q:
10            q[t] = 'v' + counter
11            counter++
12            return q[t]
13        else:
14            new_children = []
15            for child in term.children:
16                new_children.append(rename(child, q, counter))
17            term.children = new_children
18            return term

```

**Listing 4.1:** Pseudo Code for Variable Renaming

The renaming of variables is often done under a larger procedure of canonization (Section 4.4) but has been split into its own procedure as the other operations performed by canonization has an effect on the structure that should be evaluated separately.

**ESBMC Program Options**

```
--variable-renaming // Renames all variables within a formula to
                  // be looked up in the cache into a
                  // standardized variable names
```

### 4.3 Variable Dependent Formulas

A simple method of splitting up a formula into smaller parts to increase the likelihood of finding cache hits for these smaller formulas, is to split the formula into variable dependent conjuncts. This process is referred to as *slicing* in green.

$$q = \emptyset \quad (4.10)$$

$$\text{gathervars}_0(c, q) = q \cup (V(c)) \quad (4.11)$$

$$\text{gathervars}_1(c, q) = \text{gathervars}(d, q) \quad \forall d \in L(c) \quad (4.12)$$

$$\text{gathervars}(c, q) = \left\{ \begin{array}{ll} \text{gathervars}_0(c, q) & V(t) \neq \text{none} \\ \text{gathervars}_1(c, q), & V(t) = \text{none} \end{array} \right\} \quad (4.13)$$

Consider the following formula  $f$

$$f = (a + b < 0) \wedge (b + c < 0) \wedge (d + e < 0) \quad (4.14)$$

It is clear the satisfiability of the formula  $f$  can be deduced by determining the satisfiability of the sliced formulas  $f_0$  and  $f_1$  since there are no overlapping variables that could have an effect on the satisfiability of the other formula.

$$f_0 = (a + b < 0) \wedge (b + c < 0)$$

$$f_1 = (d + e < 0)$$

$$f = f_0 \wedge f_1$$

Two formulas  $f_0$  and  $f_1$  are considered to be variable dependent when the following holds:

$$\text{vardependent}(f_0, f_1) = ((q_0 \cap q_1) = \emptyset) | \text{gathervars}(f_0, q_0), \text{gathervars}(f_1, q_1) \quad (4.15)$$

$$f' = f_0 \wedge f_1 \wedge \dots \wedge f_{n-1}, \text{vars}_i \cap \text{vars}_j = \emptyset \forall i, j, i \neq j \quad (4.16)$$

The most efficient way to split clauses from formula  $f$  into smaller formulas that are variable dependent would be to make use of a *union-find* data structure. The most efficient variation being *weighted union-find with path compression*. Listing 4.2 shows the pseudo code for these operations.

**44 CHAPTER 4. FORMULA MODIFICATIONS FOR INCREASED REUSE POTENTIAL**

```

1  formulas findVariableDependant(clauses):
2      union_find
3      formulas = []
4
5      for clause in clauses:
6          variables = findVariables(clause)
7          var1 = variables[0]
8          for var2 in variables:
9              union_find.map(var1, var2)
10
11     for clause in clauses:
12         variables = findVariables(clause)
13         var1 = variables[0]
14         root = union_find.root(var1)
15         formulas[root].add(clause)
16
17     return formulas

```

**Listing 4.2:** Pseudo Code for Finding Variable Dependent formulas**Note**

As soon as there is an ITE branch, all variables from both parts of the branch are variable dependent since they will share a common variable that the branch check will be performed upon.

**ESBMC Program Options**

```
--variable-dependant // Builds the formula to be looked up
// containing only terms that are
// dependent on the related assert
```

**4.4 Canonization**

Often times part of the canonization process of a formula is to rearrange the clauses of said formula. This process cannot be replicated with the current version of the cache due to the process of subset matching in Section 3.7. Some of the filters that are used during this process assumes that the terms encountered in the formula are in a certain order to quickly eliminate terms that should not be matching. Only small adjustments will be made to ensure that the formula produced stays consistent across runs and iterations. For now, only equality signs will be forced into a canonical form and a more detailed evaluation needs to be performed to ascertain the reuse potential of a more standardized form of canonization.

## ESBMC Program Options

```
--canonize // Restructure each formula to be looked up
           // into a canonical form
```

## 4.5 Variable Propagation

### 4.5.1 Single assignment

A common strategy to increase the likelihood of matching formulas, or to decrease the amount of time taken to match two possible terms within a formula is to decrease the number of terms and making each term more unique by propagating the values of variables through the formula.

The following table shows how a formula is changed when the variables are propagated. The second formula can now arguably be matched better since each term now no longer relies on the values of other terms.

(= a 1)	(= 1 1)
(= b (+ a a))	(= b (+ 1 1))
(= c 1)	(= 1 1)
(= d 1)	(= 1 1)
(= e 1)	→ (= 1 1)
(= f 1)	(= 1 1)
(= g 1)	(= 1 1)
(= h (+ c d))	(= h (+ 1 1))
(= i (+ d e))	(= i (+ 1 1))
(= j (+ g g))	(= j (+ 1 1))

The variables encountered in this example was quite simple to propagate as each variables had an exact value and only a single assignment. The next section will explore formulas with multiple assignments to a single variable.

### 4.5.2 Multiple assignments

When propagating a single assignment, it is quite clear how to go about modifying the formula, although when there are multiple assignments more care needs to be taken. Given the following example, in which order does the assignment for variable *f* need to be propagated?

```
(= e 1)
(= a e)
(= b e)
(= c e)
(= d e)
(= f (+ a b))
(= g (+ c d))
(= f g)
```

A top down approach would yield while a bottom up approach would yield:

**46 CHAPTER 4. FORMULA MODIFICATIONS FOR INCREASED REUSE POTENTIAL**

(= e 1)	(= e 1)	
(= a 1)	(= a e)	
(= b 1)	(= b e)	
(= c 1)	→	(= c e)
(= d 1)	(= d e)	
(= f (+ 1 1))	(= f g)	
(= g (+ 1 1))	(= g (+ c d))	
(= (+ 1 1) (+ 1 1))	(= f g)	

The second propagation did not yield any valuable insight and has lost some information as to the variable  $f$ . Variable propagation will be handled top down and only on variables that are assigned to another value once. If the process of Variable Simplification (Section 4.6) was used the formula would first be modified as follows and then result in the final propagated formula where the order is no longer important.

(= e 1)	(= 1 1)	
(= e e)	(= 1 1)	
(= e e)	(= 1 1)	
(= e e)	→	(= 1 1)
(= e e)	(= 1 1)	
(= g (+ e e))	(= g (+ 1 1))	
(= g (+ e e))	(= g (+ 1 1))	
(= g (+ e e))	(= g (+ 1 1))	

**ESBMC Program Options**

```
--variable-propagation // Propagates all variable values
```

**4.6 Variable Simplification**

To help out Section 4.5, as many simplifications will be made as possible with regards to variables equaling other variables.

```
(= a 1)
(= b 2)
(= c a)
(= c b)
```

We use union find on all top-level equalities between 2 variables. In this case  $a$ ,  $b$  and  $c$  all share the same root, in this case  $a$ . Each variable is replaced by its root. It does not matter if we introduce an obvious contradiction, as the contradiction would have been found by the solver in any case (the solver still finds it with slightly different names now).

```
(= a 1)
(= a 2)
(= a a)
(= a a)
```

We now also have fewer variables and complications to keep track of during the propagation process of the previous section and no complex or nested dependencies.

## ESBMC Program Options

```
--variable-simplification // Simplifies all variables that are
// assigned to each other
```

### Splitting Variable Propagation and Variable Simplification into separate functions

Section 4.6 and 4.5 is split into separate sections and options, since they can have very different effects on the formulas, mainly to do with the option of solving the lookup. Variable propagation can cause a formula to become extremely large where it was previously quite compact, and each time it is found will have to be re-processed by the solver (unless internal caching was able to identify it and reuse some of the knowledge, provided the other variables didn't also change along the way).

```
= a 1
= b (+ a a)
= c (+ b b)
= d (+ c c)
= e (+ d d)
```

Doing variable propagation would cause the formula to swell to the following

```
= a 1
= b (+ 1 1)
= c (+ (+ 1 1) (+ 1 1))
= d (+ (+ (+ 1 1) (+ 1 1)) (+ (+ 1 1) (+ 1 1)))
= e (+ (+ (+ (+ 1 1) (+ 1 1)) (+ (+ 1 1) (+ 1 1))) ...)
```

Although in this trivial example, further simplifications can be done to create a single constant (explored further in Section 4.7). Some examples where the variables aren't constants will stay large and quite complex.

## 4.7 Formula Simplification

There are multiple approaches when it comes to simplifying the formula during processing. Three main approaches will be discussed in this section namely, simplify, simplify during and simplify replaced.

### 4.7.1 Simplify Replaced

When a variable is propagated or simplified in section 4.6 or 4.5 and the new term contains a formula that is being assigned to itself as seen in the following example:

```
(= a a)
(True)
```

This term can be removed from the list of conjuncts. Leaving in the conjunct will have no effect on the subset lookup but will have an effect on looking up exact formulas. It is a very inexpensive process to potentially gain during exact lookup.

## ESBMC Program Options

```
--simplify-replaced // Simplifies terms where
// both sides of the assignment
// are the same
```

### 4.7.2 Simplify

Adding the simplify option as a parameter, will look at all parts of the formula and attempt to do any simplification based on a set of rules stated in ESBMC. This includes rules like simplifying clauses to True or False, removing True clauses from conjuncts, removing False clauses from Disjuncts, removing double negation. This process can be quite time-consuming as all parts of the formula need to be traversed and tested against the stated rules.

## ESBMC Program Options

```
--simplify // Simplifies terms as much as possible with
// rules specified internally by ESBMC
```

### 4.7.3 Simplify During

Similar to doing simplification, but will preemptively simplify a variable assignment before it gets propagated to avoid having the same subexpression evaluated multiple times. This option also uses caching to see if any subexpression has been evaluated before.

```
--simplify-during // Simplifies terms as much as possible with
// rules specified internally by ESBMC
// while propagating values
```

## 4.8 Splitting Assumptions From Asserts

Examining how asserts are being handled again by Section 3.2, there is a modification done to the *assert* before it is sent to the SMT solver. The assert produced by the SSA Step is combined with all the assumptions that have been encountered up to that point.

$$c = \neg(assumptions \Rightarrow assert) \quad (4.17)$$

$$assumptions = c_0 \wedge c_1 \wedge \dots \wedge c_{n-1} \quad (4.18)$$

Assumptions are already in the form of a conjunct and if they were to be added to the list of clauses of the formula  $f$  then there is a bigger chance of finding a subset match, as only some of the assumptions would have had to be present during previous iterations, as is currently the case for assignments.

The statement  $a \Rightarrow b$  can be rewritten as  $\neg a \vee b$ , and applying it to the original construction of the clause to be added to the formula when an assert is found, produces a new structure.

$$\neg(assumptions \Rightarrow assert) \quad (4.19)$$

$$\neg(\neg assumptions \vee assert) \quad (4.20)$$

$$assumptions \wedge \neg assert \quad (4.21)$$

The assumptions can now be integrated as part of the regular assignments. The end effect being the assert has been simplified for lookup as well as more possible terms to match with by combining the assumptions into the list of asserts.

```
--split-assumptions-and-assert // splits the assumptions
                                // from the assert term
                                // and adds them to the list
                                // of assignments
```



## Chapter 5

# Evaluating Cache Performance

### 5.1 Experimental Setups and Overview

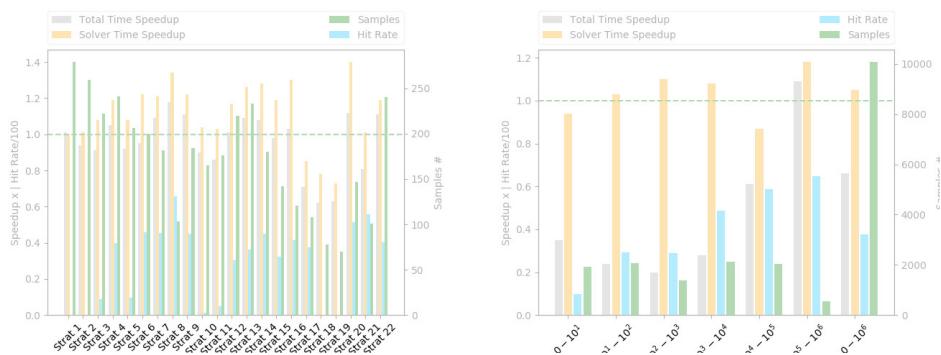
To evaluate the impact the introduction of formula caching has had on ESBMC, a collection of different caching strategies will be run in ESBMC with iterative deepening over three different sets of benchmarks over two different sets of hardware specifications. The first set of benchmarks (referred to as data set 1) will be that of SV-COMP 2016 to be run with small timeouts and the *Z3 SMT Solver* to determine the best strategies. Dataset will not include the data related to multi-threading, as that data will be evaluated separately in data set 2. A selection of strategies will then be run on a cluster with larger timeouts over the current SV-COMP 2017 benchmarks with both Z3 (referred to as data set 2) and boolector (referred to as data set 3) as the SMT solvers to see if the first set of results correlated to the results found on the clusters which are regularly used to evaluate the performance of ESBMC as well as the overall effectiveness of the formula cache. Throughout the evaluation, the baseline will refer to the results obtained from running ESBMC without caching activated but with all other options exactly the same as to the strategy with which it is being compared. The terms *strategy* and *experiment* will also be used interchangeably with *experiment x* referring to the *strategy x* being applied to a specific data set. It is expected that the main effect of caching will be visible when it comes to evaluating the speedup of the SMT solvers as the smaller formulas (if cache hits were registered) should decrease the amount of work required from the SMT solver. Since there is an inherent overhead with performing the storing and retrieval of results in the cache it can be expected that the most gains are seen in the categories where the problems are hard (take a substantial amount of time to solve) for the SMT solvers to return a satisfiability result. For evaluations where the cache can return a result within a few milliseconds, it is expected for the overhead to dominate any increase in performance that cache hits may offer. For problems specifically related to multi-threading it is expected to see a great deal of reuse as the same problems should be appearing multiple times within a single iteration.

Appendix A gives detailed results for each strategy that is run on data set 1, the conclusions from these data sets are discussed during the relevant evaluation but are added to provide extra insight if so required by the reader. Throughout the chapter the column headings will contain plenty of abbreviated terms, please refer to the Glossary for further information.

## 5.2 Interpreting data from graphs

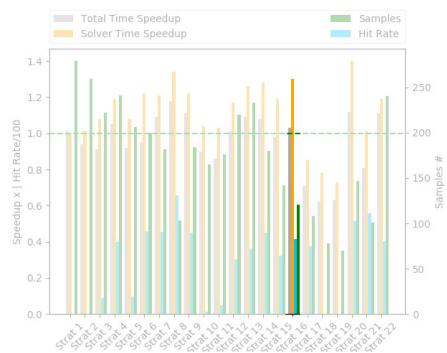
The number of different data points generated by the experiments run for the purpose of evaluating the cache is quite large and presenting these results in a compact but insightful way has lead to graphs with lots of combined data. This section will elaborate as to what each part of the graphs represents and how to quickly infer information from the data that is displayed. During the evaluations where a graph appears, there will be an accompanying table that will give exact figures and units of measurement of the relative graph.

### Different Experiments



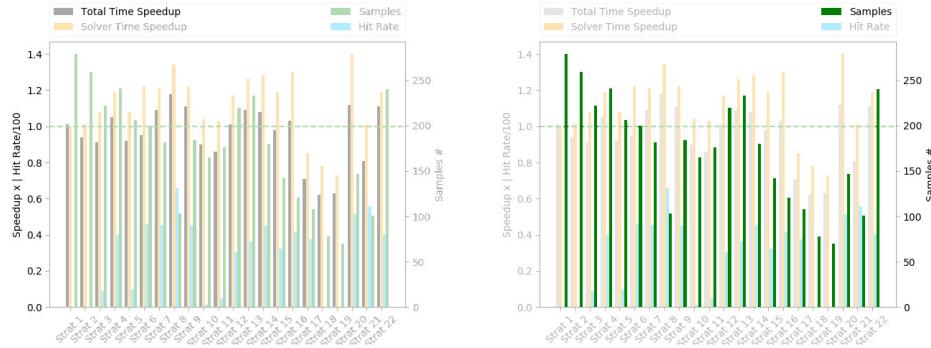
During the evaluations, there are mainly two different evaluations, one which takes a given time range for the baseline solver time and pulls samples from the data of each strategy that completed the same data points as the baseline. These experiments have a set of strategies as the labels for the x-axis. The second evaluation takes a strategy and displays data points for which the baseline also completed and clusters these ranges of baseline solver time with their representative data points as the labels for the x-axis.

### Clusters



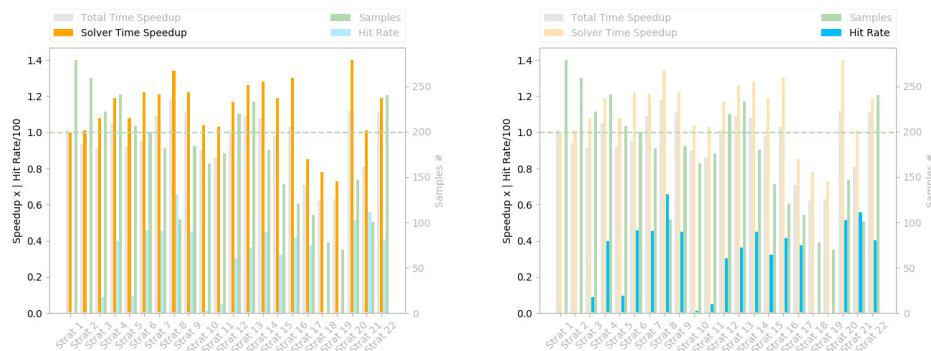
For each label on the x-axis, a cluster of results is displayed for the specific label. For the larger graphs, the gaps between the clusters can become quite small. Each cluster contains four bars and represents (from left to right) the total time speedup (gray bar), the solver time speedup (orange bar), the hit rate (blue bar) and the number of samples (green bar).

## Total Time Speedup and Samples



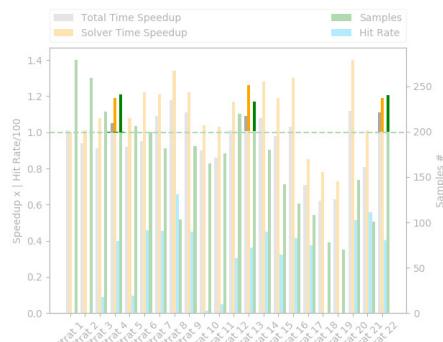
The total time speedup and the number of samples completed are a good indication as to the success of goal number 1 as a total time speedup of more than 1 along with a good amount of samples indicate the success of goal 1.

## Solver Time Speedup Hit Rate



The solver time speedup and the remove rate are good indications of goal number as a solver time speedup of more than 1.0 indicate the solver is getting faster with smaller formulas to be solved and the hit rate indicating how often the formula is made smaller.

## Good results



When reading the graphs, a good rule of thumb is that a strategy with three of its four (the hit rate cannot go above the line) bars above the green line is performing really well as they have achieved a net increase in performance with regards to the SMT solver, ESBMC and have a good amount of samples to give a good amount of confidence in the result.

## 5.3 Strategies

The following table explains the options that are enabled for the cache when the corresponding set of experiments are run for a given strategy.

### 5.3. STRATEGIES

55

Option	Strat	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Exact-Lookup (Section 3.6)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Combine Asserts (Section 3.3)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Variable Renaming (Section 4.2)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Expression Relevant (Section 4.1)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Variable Dependent (Section 4.2)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Canonization (Section 4.4)																							
Variable Simplification (Section 4.6)																							
Variable Propagation (Section 4.5)																							
Simplify Replaced (Section 4.7.1)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Simplify (Section 4.7.2)																							
Store During Lookup (Section 3.9)																							
Subset-Lookup (Section 3.7)																							
Rename Filter (Section 3.7.4)																							
Shape Filter (Section 3.7.4)																							
Range Filter (Section 3.7.4)																							
Forward Backward Filter (Section 3.7.4)																							
Split Assumptions Assert (Section 4.8)																							
Unsat-Cone-Lookup (Section 3.8)																							

**Table 5.1:** Overview of strategies and their relevant options

## 5.4 Data Set 1

### 5.4.1 Setup

#### SMT Solver: Z3

For data set 1 an Intel 5820k 6 core multi-threaded CPU at 3.3 GHz with 16GB of RAM will be used to perform all experiments on. Each run was limited to 3 minutes and with a 2GB memory cap to get initial estimates before larger experiments were run on a cluster. All verification runs were made across the SV-COMP 2016 data set with the following parameters along with the parameter specified in the setup of the experiment in their relevant sections. Each verification run starts with an empty cache. For data set 1 the samples containing concurrency data have been excluded (see data set 2 (Section 5.5) for their results). The SMT solver used during this data set is Z3.

```
--bv --z3 --falsification --timeout 180 --memlimit 2g
```

The evaluations for data set 1 is split up into 4 different groups that fall under the following categories: lookup, subset-lookup, unsat-core-lookup and hybrids. These 4 categories evaluate caching strategies that are specific to them. Section 5.4.3 through Section 5.4.6 analyses the results from experiments that only make use of *Exact Matches* (Section 3.6). Section 5.4.7 through Section 5.4.8 specifically evaluates the effectiveness that *subset-lookup* (Section 3.7) has on verification runs for sample files found in data set 1. Unsat-core-lookups are evaluated in Section 5.4.9 and finally hybrid strategies in Section 5.4.10.

### 5.4.2 Overview of results

#### Experiments where the baseline had a solving time between 0s and 1000s

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	83.32	0.94	-26.28	1.02	10.53	13280
2	475.81	0.72	-32.08	1.03	10.67	13111
3	579.60	0.64	-132.90	1.17	24.67	12883
4	275.87	0.80	-212.91	1.27	39.85	13101
5	884.00	0.52	-129.59	1.17	25.00	12676
6	1003.18	0.48	-190.91	1.28	40.01	12703
7	474.47	0.64	-159.04	1.26	40.61	12312
8	2158.57	0.27	-231.26	1.48	56.31	8096
9	413.26	0.67	-163.21	1.27	40.70	12299
10	1682.55	0.34	-19.08	1.03	18.24	11554
11	1698.45	0.35	-39.91	1.05	24.65	11719
12	754.42	0.59	-182.43	1.25	39.50	12917
13	490.59	0.69	-234.72	1.33	41.20	12985
14	633.01	0.58	-179.92	1.29	43.70	12426
15	1059.82	0.41	-125.97	1.23	43.20	12242
16	1514.60	0.30	-137.83	1.29	45.65	12040
17	1831.98	0.27	123.09	0.82	31.97	11370
18	2117.02	0.19	354.51	0.55	0.07	11473
19	2051.20	0.18	358.42	0.54	0.00	11363
20	1185.88	0.38	-189.12	1.38	46.51	12198
21	1690.64	0.25	31.36	0.94	47.45	11820
22	207.65	0.84	-210.50	1.28	39.84	13082

**Table 5.2:** Performance of all strategies where the baseline had a solving time between 0s and 1000s for data set 1

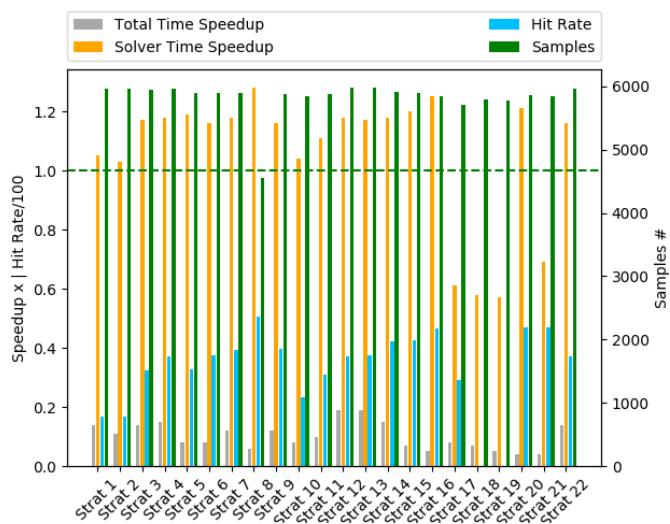


**Figure 5.1:** Performance of all strategies where the baseline had a solving time between 0s and 1000s for data set 1

### Experiments where the baseline had a solving time between 0ms and 10ms

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	267.25	0.14	-0.13	1.05	16.89	5963
2	335.33	0.11	-0.08	1.03	16.91	5956
3	411.59	0.14	-0.34	1.17	32.44	5950
4	396.98	0.15	-0.37	1.18	37.15	5955
5	699.09	0.08	-0.39	1.19	32.73	5899
6	730.91	0.08	-0.33	1.16	37.52	5898
7	450.59	0.12	-0.37	1.18	39.46	5887
8	2135.34	0.06	-0.40	1.28	50.55	4560
9	440.06	0.12	-0.33	1.16	39.54	5884
10	625.41	0.08	-0.10	1.04	23.20	5850
11	562.63	0.10	-0.23	1.11	31.04	5875
12	397.98	0.19	-0.36	1.18	37.34	5976
13	397.23	0.19	-0.35	1.17	37.44	5976
14	448.31	0.15	-0.37	1.18	42.35	5905
15	488.35	0.07	-0.40	1.20	42.56	5890
16	684.22	0.05	-0.47	1.25	46.74	5845
17	870.83	0.08	1.53	0.61	29.11	5714
18	676.32	0.07	1.73	0.58	0.15	5801
19	702.63	0.05	1.78	0.57	0.00	5775
20	594.78	0.04	-0.42	1.21	46.95	5861
21	670.20	0.04	1.10	0.69	46.93	5850
22	409.62	0.14	-0.34	1.16	37.16	5955

**Table 5.3:** Performance of all strategies where the baseline had a solving time between 0ms and 10ms for data set 1



**Figure 5.2:** Performance of all strategies where the baseline had a solving time between 0ms and 10ms for data set 1

### Experiments where the baseline had a solving time between 11ms and 100ms

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	17.03	0.87	-0.34	1.01	2.74	3032
2	182.74	0.33	-0.99	1.03	2.75	3018
3	330.47	0.19	-5.11	1.15	10.92	2994
4	183.28	0.32	-4.04	1.11	25.11	3011
5	510.37	0.13	-5.29	1.15	11.09	2948
6	662.84	0.10	-3.62	1.10	25.35	2952
7	446.85	0.13	-4.67	1.14	24.91	2922
8	3016.86	0.02	-7.97	1.26	41.34	1401
9	429.58	0.13	-4.52	1.13	25.03	2922
10	1792.46	0.05	-2.76	1.08	9.25	2859
11	1862.88	0.04	-4.92	1.14	13.52	2897
12	526.95	0.15	-4.68	1.14	27.03	3020
13	289.88	0.24	-6.47	1.20	30.54	3020
14	482.38	0.12	-6.59	1.20	30.53	2934
15	671.47	0.09	-6.60	1.20	30.65	2933
16	902.03	0.07	-7.57	1.24	31.38	2900
17	1697.49	0.04	66.49	0.36	21.12	2769
18	1370.40	0.04	81.57	0.32	0.00	2862
19	1270.41	0.05	82.05	0.32	0.00	2845
20	818.79	0.07	-7.75	1.25	31.44	2904
21	1438.72	0.04	56.29	0.41	32.07	2830
22	134.92	0.38	-4.40	1.13	25.05	3008

**Table 5.4:** Performance of all strategies where the baseline had a solving time between 11ms and 100ms for data set 1

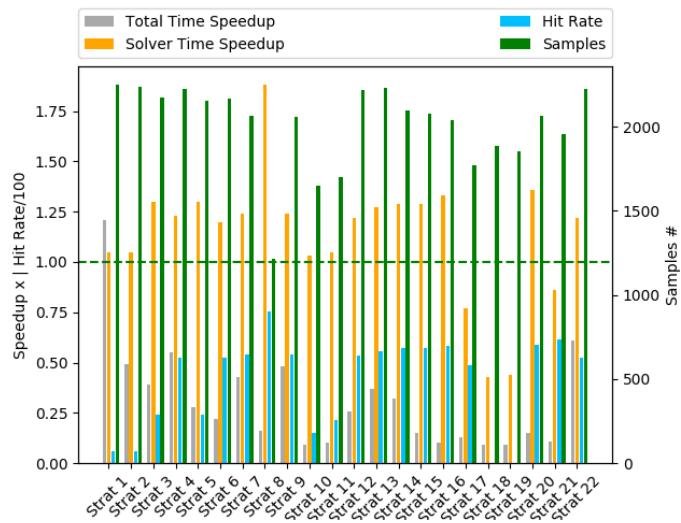


**Figure 5.3:** Performance of all strategies where the baseline had a solving time between 11ms and 100ms for data set 1

### Experiments where the baseline had a solving time between 101ms and 1000ms

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	-116.72	1.21	-17.21	1.05	6.18	2248
2	651.81	0.49	-16.46	1.05	6.21	2239
3	692.37	0.39	-83.03	1.30	23.93	2173
4	373.26	0.55	-67.38	1.23	52.57	2224
5	1086.14	0.28	-82.45	1.30	24.14	2154
6	1533.37	0.22	-60.51	1.20	52.56	2164
7	530.85	0.43	-68.16	1.24	53.81	2065
8	1940.20	0.16	-158.67	1.88	75.27	1215
9	444.89	0.48	-68.63	1.24	53.88	2060
10	5150.13	0.09	-9.82	1.03	15.14	1647
11	5201.57	0.10	-16.33	1.05	21.38	1698
12	1869.03	0.26	-65.51	1.22	53.61	2219
13	1069.82	0.37	-76.61	1.27	55.90	2230
14	881.01	0.32	-79.58	1.29	57.42	2093
15	2384.14	0.15	-80.71	1.29	57.33	2076
16	3488.70	0.10	-87.73	1.33	58.07	2037
17	3197.90	0.13	107.17	0.77	48.97	1772
18	4030.51	0.09	462.56	0.43	0.00	1888
19	4114.89	0.09	460.32	0.44	0.00	1855
20	2294.66	0.15	-93.28	1.36	58.62	2065
21	3328.53	0.11	59.29	0.86	61.33	1953
22	292.67	0.61	-65.63	1.22	52.57	2221

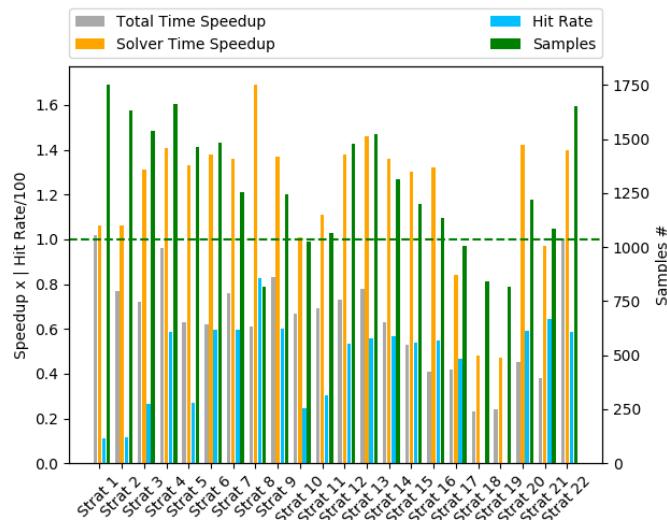
**Table 5.5:** Performance of all strategies where the baseline had a solving time between 101ms and 1000ms for data set 1



**Figure 5.4:** Performance of all strategies where the baseline had a solving time between 101ms and 1000ms for data set 1

**Experiments where the baseline had a solving time between 1s and 10s**

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	-81.72	1.02	-164.12	1.06	10.91	1751
2	1088.28	0.77	-180.68	1.06	11.70	1633
3	1238.32	0.72	-718.48	1.31	26.45	1539
4	154.80	0.96	-904.24	1.41	58.78	1664
5	1841.24	0.63	-736.77	1.33	27.17	1465
6	1981.79	0.62	-840.92	1.38	59.51	1485
7	941.81	0.76	-778.11	1.36	59.85	1254
8	1921.90	0.61	-1197.51	1.69	82.53	816
9	636.79	0.83	-788.42	1.37	60.22	1247
10	1627.89	0.67	-20.48	1.01	24.46	1027
11	1462.48	0.69	-267.70	1.11	30.31	1067
12	1237.40	0.73	-802.89	1.38	53.62	1476
13	937.17	0.78	-919.69	1.46	55.62	1521
14	1761.15	0.63	-752.58	1.36	56.87	1312
15	2577.19	0.53	-641.32	1.30	53.75	1199
16	4055.01	0.41	-646.58	1.32	55.06	1135
17	4315.03	0.42	506.96	0.84	46.63	1004
18	9109.86	0.23	2847.01	0.48	0.00	843
19	8654.03	0.24	2943.56	0.47	0.00	816
20	3495.52	0.45	-820.83	1.42	59.38	1219
21	4563.06	0.38	90.40	0.97	64.44	1085
22	-5.43	1.00	-888.97	1.40	58.81	1653

**Table 5.6:** Performance of all strategies where the baseline had a solving time between 1s and 10s for data set 1**Figure 5.5:** Performance of all strategies where the baseline had a solving time between 1s and 10s for data set 1

### Experiments where the baseline had a solving time between 10s and 100s

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	-346.15	1.01	-120.78	1.00	0.00	279
2	1731.40	0.94	-316.22	1.01	0.00	259
3	2499.88	0.91	-1854.77	1.08	9.01	222
4	-1237.95	1.05	-4119.22	1.19	39.82	241
5	2216.00	0.92	-1827.12	1.08	9.71	206
6	1414.51	0.95	-4495.44	1.22	45.93	200
7	-2204.90	1.09	-4535.45	1.21	45.54	182
8	-3089.08	1.18	-6666.22	1.34	65.64	103
9	-2562.82	1.11	-4647.27	1.22	45.04	184
10	3022.02	0.90	-918.74	1.04	1.12	165
11	4540.16	0.86	-727.98	1.03	5.04	176
12	-246.32	1.01	-3912.56	1.17	30.30	220
13	-2223.09	1.09	-5499.85	1.26	36.12	233
14	-1969.48	1.08	-5889.58	1.28	44.97	180
15	446.18	0.98	-4260.15	1.19	32.25	142
16	-791.08	1.03	-5992.23	1.30	41.59	121
17	10310.27	0.71	4341.38	0.85	37.58	108
18	14403.05	0.62	6713.95	0.78	0.00	78
19	13112.10	0.63	7967.07	0.73	0.00	70
20	-2742.86	1.12	-7391.95	1.40	51.19	147
21	5090.55	0.81	-256.62	1.01	55.85	101
22	-2617.89	1.11	-4041.97	1.19	40.35	240

**Table 5.7:** Performance of all strategies where the baseline had a solving time between 10s and 100s for data set 1



**Figure 5.6:** Performance of all strategies where the baseline had a solving time between 10s and 100s for data set 1

**Experiments where the baseline had a solving time between 100s and 1000s**

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	-5239.14	1.04	1780.43	0.99	0.00	7
2	763.17	0.99	-555.83	1.00	0.00	6
3	12680.60	0.91	613.40	1.00	0.00	5
4	-15177.83	1.14	-21291.33	1.21	2.50	6
5	20926.25	0.86	2124.50	0.98	0.00	4
6	-17070.25	1.15	-33432.00	1.35	3.75	4
7	3703.50	0.98	-148.00	1.00	0.00	2
8	-2725.00	1.02	-2732.00	1.02	0.00	1
9	-413.00	1.00	-6313.00	1.04	0.00	2
10	683.00	0.99	-3871.17	1.03	0.00	6
11	1683.83	0.99	-1759.67	1.01	0.00	6
12	-24068.17	1.24	-24825.17	1.25	2.50	6
13	-34435.40	1.37	-34995.80	1.39	3.80	5
14	1522.50	0.99	-64.50	1.00	0.00	2
15	12817.50	0.92	10530.50	0.93	0.00	2
16	3703.50	0.98	1460.00	0.99	0.50	2
17	13616.00	0.90	12966.00	0.90	0.00	3
18	30525.00	0.81	26800.00	0.83	0.00	1
19	11548.50	0.92	7759.00	0.94	0.00	2
20	2659.00	0.98	-1023.00	1.01	0.50	2
21	25436.00	0.84	17062.00	0.89	1.00	1
22	-28276.60	1.29	-30645.40	1.32	3.00	5

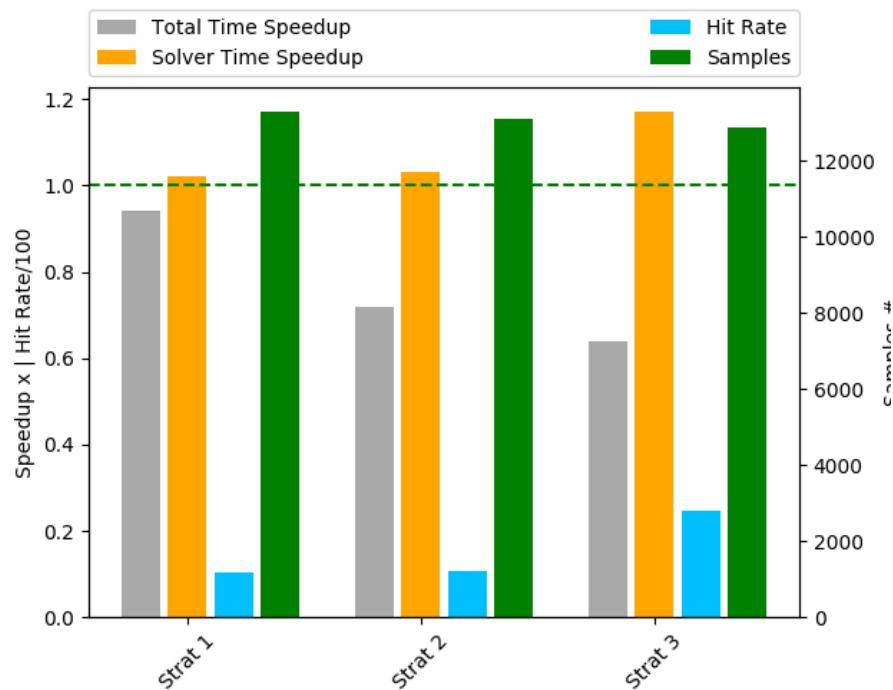
**Table 5.8:** Performance of all strategies where the baseline had a solving time between 100s and 1000s for data set 1**Figure 5.7:** Performance of all strategies where the baseline had a solving time between 100s and 1000s for data set 1

### 5.4.3 No modifications and Variable Renaming

Detailed data can be found for strategy 1 on page 122, strategy 2 on page 125 and strategy 3 on page 128.

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	83.32	0.94	-26.28	1.02	10.53	13280
2	475.81	0.72	-32.08	1.03	10.67	13111
3	579.60	0.64	-132.90	1.17	24.67	12883

**Table 5.9:** Performance of strategy 1, 2 and 3 for all samples in data set 1



**Figure 5.8:** Performance of strategy 1, 2 and 3 for all samples in data set 1

Experiment 1 and experiment 2 are in essence the same experiment with the only difference being that during experiment 2, all the asserts were handled individually instead of being handled as one large assert like experiment 1. These are the most basic options for the cache to start operating and other experiments will build on each other to slightly modify the way the cache behaves during each experiment. When examining the results from experiment 1 and experiment 2, inconsistencies can be seen with regards to the hit rate of these strategies. This variation occurs because of the different amount of samples that each experiment completed. When considering the samples where both completed the same hit rate is observed for both. When the two

experiments are compared with regards to the  $\Delta\text{TT Avg}$  a larger difference in times appear across all the different baseline solver time ranges. These discrepancies in times are due to the way that experiment 2 is storing and retrieving formulas as it is doing a lot more cache lookups and inserts, as each assert is given its own formula to interact with the cache compared to experiment 1 where all asserts are combined into a single formula to interact with the cache.

It could have been expected for experiment 2 to gain in hit rate, as there are more formula lookups and in some cases, it could be expected that the same formula appears in subsequent unwindings, even if the list of assignments should grow for each subsequent unwinding. For the most part, experiment 1 and experiment 2 perform relatively equally with regards to  $\Delta\text{ST Avg}$ , with some exceptions where the number of samples are different, as could be expected.

Further investigating the formulas sent to the SMT solver reveals a mechanism that is built into ESBMC with regards to the naming of non-determinant variables. Consider the following c code and the related formula sent to the SMT solver for the unwind bound  $k = 1$ .

```
int x;
int y = x;
assert (x != y);

(= y nondet1)
(= x y)
```

Ignoring the completeness threshold, and creating the formula to be sent to the SMT solver for unwind bound  $k = 2$  produces the following:

```
(= y nondet2)
(= x y)
```

The variable  $x$  was not assigned an explicit value, ESBMC thus marks it as a symbolic variable for which the value is non-determinate while assigning it a new name. When the variable is assigned a name, it is assigned a suffix from a counter as well. This counter does not reset during subsequent unwindings, making it nearly impossible to find cache hits. As a result of this mechanism, there should never be a case where there were more cache hits in experiment 2 compared to experiment 1.

Renaming formulas appear to be a crucial criterion (as seen by *Green* as well) to gain any significant amount of reuse from a formula cache. Experiment 3 builds on experiment 2 with the added options of *Variable Renaming* (Section 4.2) applied to the formulas. The results of experiment (Section A.3) show a clear increase in hit rate across all the baseline time ranges when compared to that of experiment 1 and experiment 2. These increases directly correlate to an increase for ST  $x$  as there was less work for the solver to do during each call. When considering  $\Delta\text{TT Avg}$  this increase in hit rates came at a substantial cost as the cache had to do more work renaming the variables before storing or retrieving formulas.<sup>1</sup>

---

<sup>1</sup>See Section 5.4.11 for more detailed analysis as to where time is lost.

**Insights**

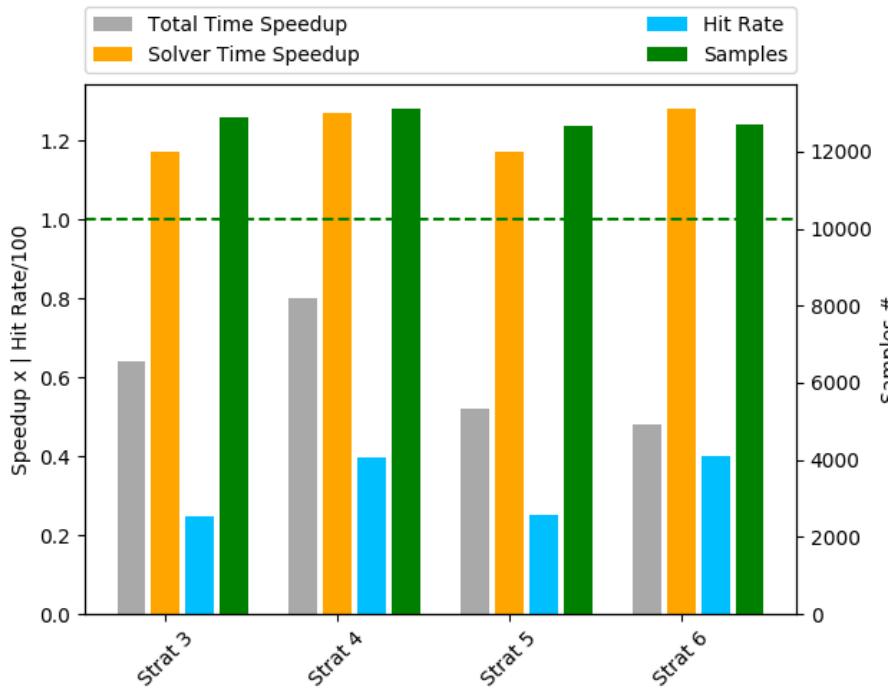
Effective caching of formulas rely heavily on the variables being renamed into a standardized form. Experiment 1 and experiment 2 also shows that the greatest amount of speedup comes from removing the entire formula and not just parts of it as the time saved is guaranteed by avoiding the solver call altogether.

#### 5.4.4 Slicing Assignments

Detailed data can be found for strategy 3 on page 128, strategy 4 on page 131, strategy 5 on page 134 and strategy 6 on page 137.

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
3	579.60	0.64	-132.90	1.17	24.67	12883
4	275.87	0.80	-212.91	1.27	39.85	13101
5	884.00	0.52	-129.59	1.17	25.00	12676
6	1003.18	0.48	-190.91	1.28	40.01	12703

**Table 5.10:** Performance of strategy 3, 4, 5 and 6 for all samples in data set 1



**Figure 5.9:** Performance of strategy 3, 4, 5 and 6 for all samples in data set 1

Experiment 4 and experiment 5 take different approaches to the same end goal of slicing assignments from the formula to be looked up and both build on experiment 3. Experiment 4 focuses on only using assignments that were present when an assert was found (Section 4.1) while experiment 5 sliced the assignments by identifying the assignments that are variable dependent to the assert (Section 4.2).

Experiment 4 had a greater hit rate compared to that of experiment 5 across all baseline solver time ranges but did not reflect a greater solver speedup all of the time. This can be attributed to experiment 5 having completed fewer examples. It can be

assumed that these were hard examples where experiment 5 timed out while affecting the measured speed up of experiment 4. For the experiments with a baseline solver time of more than 1s there appears a significant performance increase with regards to the solver time speedup. The more important metric to look at when comparing these experiments (which is also the cause of fewer samples to look at) is  $\Delta TT$  Avg. Experiment 5 is significantly slower when considering the total time that each verification runs takes on average. This is due to the large amount of time it takes to find the variable dependent parts of each formula. Even though the most efficient implementation of union-find is used, it is still costly to explore a formula for each assert that is being looked up.

Experiment 6 (Section A.6) combines the approaches from experiment 4 and experiment 5. Combining the strategies from experiment 4 and experiment 5 into experiment 6 hardly made any difference with an increase in the hit rate only slightly higher than that found for experiment 4 across all the baseline ranges but with sample sizes closer to that of experiment 5 than to experiment 4, indicating the time to find the variable dependent sets are still the dominating factor. The increase in hit rate came at a substantial cost compared to experiment 4 and only found some success for the samples where the baseline solver time was more than 100s, for which there are very few samples to give much confidence.

### Insights

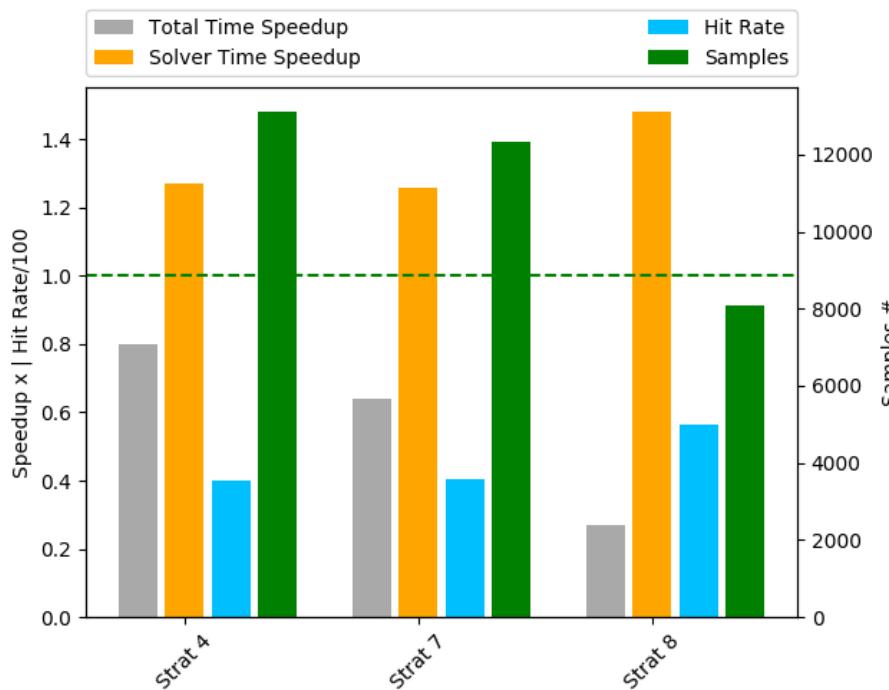
It can safely be concluded that the simple approach of experiment 4 which only considers *assignments* found when an *assert* is present is by far the most effective way of slicing assignments thus far. Combining the strategies from experiment 4 and experiment 5 into experiment 6 did yield an increase in hit rate but the inherent cost from finding the variable dependent assignments does not yield a performance increase with regards to the total time speedup.

### 5.4.5 Formula Modifications

Detailed data can be found for strategy 4 on page 131, strategy 7 on page 140 and strategy 8 on page 143.

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
4	275.87	0.80	-212.91	1.27	39.85	13101
7	474.47	0.64	-159.04	1.26	40.61	12312
8	2158.57	0.27	-231.26	1.48	56.31	8096

**Table 5.11:** Performance of strategy 4, 7 and 8 for all samples in data set 1



**Figure 5.10:** Performance of strategy 4, 7 and 8 for all samples in data set 1

Experiment 7 and experiment 8 build on experiment 4 and focuses on modifying the remaining terms through Canonization (Section 4.4), Variable Simplification (Section 4.6) and Variable Propagation (Section 4.5). The difference between the approach taken by experiment 7 and experiment 8 is the way in which the remaining terms are simplified (Section 4.7) after the modifications have taken place. Experiment 7 only simplifies the terms that were propagated (Section 4.7.1) while experiment 8 takes the more computationally expensive route of attempting to simplify as much of the structure from all the remaining terms (Section 4.7.2) as possible.

Both experiment 7 and experiment 8 are experiencing an increase in hit rate compared to experiment 4, with experiment 8 especially seeming to perform well. When examining the number of samples each experiment has, a substantial difference can be observed between experiment 8 and the other two. It can be concluded that experiment 8 performs exceptionally well when there is enough time to perform the required operations on the formulas with regards to the number of formulas that can be found in the cache. It does come at a substantial cost as these modifications are expensive. When considering the  $\Delta TT\text{ Avg}$  it is clear that experiment 8 is taking a long time to perform the relevant simplifications even though they do give an increase in hit rate.

### **Insight**

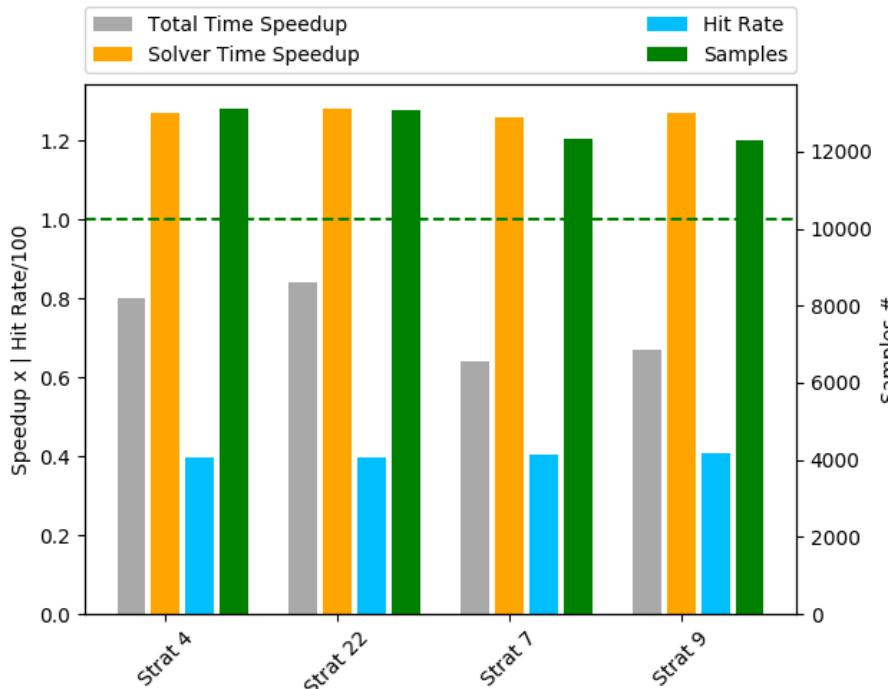
It can safely be concluded that limited simplification implemented in experiment 7 offers the best performance for the amount of time spent compared to experiment 8, but neither approach is performing better than experiment 4 when considering the average total time delta between the experiments and the baseline.

### 5.4.6 Storing formulas before their satisfiability is known

Detailed data can be found for strategy 4 on page 131, strategy 7 on page 140, strategy 9 on page 146 and strategy 22 on page 185.

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
4	275.87	0.80	-212.91	1.27	39.85	13101
22	207.65	0.84	-210.50	1.28	39.84	13082
7	474.47	0.64	-159.04	1.26	40.61	12312
9	413.26	0.67	-163.21	1.27	40.70	12299

**Table 5.12:** Performance of strategy 4, 22, 7 and 9 for all samples in data set 1



**Figure 5.11:** Performance of strategy 4, 22, 7 and 9 for all samples in data set 1

Section 3.9 describes a method of storing results before they are known by assuming that the result would be unsatisfiable. Experiment 9 implements this feature on top of experiment 7 and experiment 22 implements it over experiment 4 . Both experiments that implemented the technique to store during lookup had a slight increase in hit rate but a slight decrease in the number of samples they were able to complete in the allocated amount of time.

### Insight

It is debatable if implementing this technique is worthwhile as the performance differences are rather small.

#### 5.4.7 Subset Filters

Detailed data can be found for strategy 10 on page 149, strategy 11 on page 152, strategy 12 on page 155 and strategy 13 on page 158.

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
10	1682.55	0.34	-19.08	1.03	18.24	11554
11	1698.45	0.35	-39.91	1.05	24.65	11719
12	754.42	0.59	-182.43	1.25	39.50	12917
13	490.59	0.69	-234.72	1.33	41.20	12985

**Table 5.13:** Performance of strategy 10, 11, 12 and 13 for all samples in data set 1



**Figure 5.12:** Performance of strategy 10, 11, 12 and 13 for all samples in data set 1

For a *subset-lookup* to be successful, the renaming filter needs to be present at the very least, without it the information to prove that there is a subset will never be sufficient. The hit rate achieved by experiment 10 was not too great at a large computational cost to determine if there exists a formula  $g$  that is a subset of the current

formula  $f$  being looked up. Experiment 11 to experiment 13 starts to introduce the remaining filters to evaluate the incremental influence of each. Experiment 11 introduces the shape filter, experiment 12 then further introduces the range filter and finally experiment 13 also introduces the forward and backward filter (Section 3.7.4). Each of these experiments builds on top of the previous one. A clear trend appears where each subsequent strategy is gaining performance in hit rate, solver time and average time across all baseline solver time ranges. These increases also stayed consistently higher with more samples available for each subsequent strategy to confirm the results. The introduction of each new filter is complementing the work of the other filters allowing for more matching of formulas as well as less time to find each match. From the data it would appear that introducing the Range Filter from Section 3.7.4 made the biggest contribution, it is also this filter that introduces the smaller amount of space needed to track the potential matches, giving the cache further performance enhancements.

### Insight

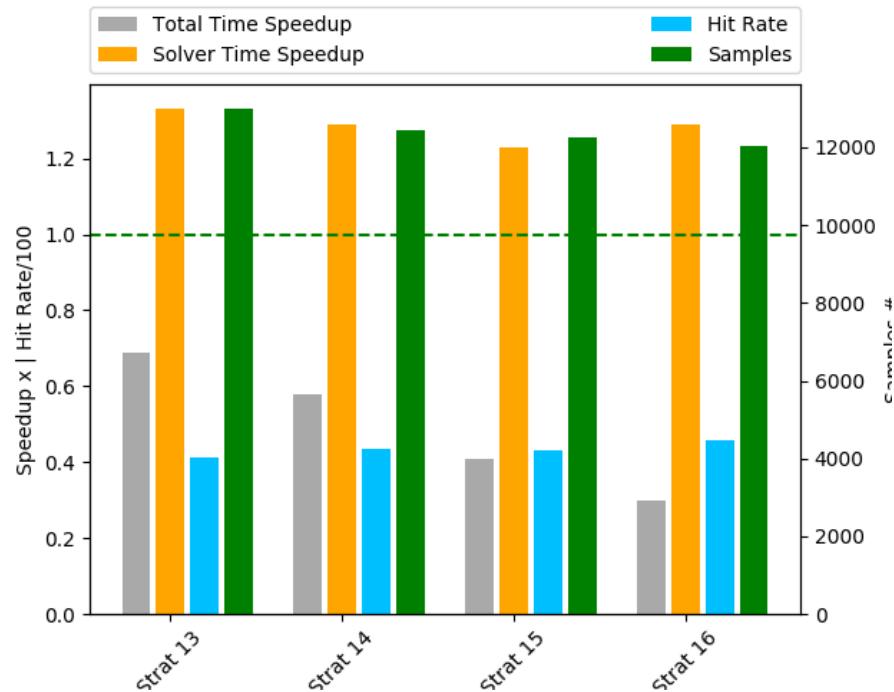
The introduction of each filter furthered strengthened the reasoning power of the subset matching process, increasing the hit rate and decreasing the amount of time it took to prove the existence (or absence) of an unsatisfiable formula in the cache that is a subset of the current formula.

### 5.4.8 Formula modifications

Detailed data can be found for strategy 13 on page 158, strategy 14 on page 161, strategy 15 on page 164 and strategy 16 on page 167.

Strat	$\Delta TT\ Avg(ms)$	$TT\ x$	$\Delta ST\ Avg(ms)$	$ST\ x$	HR(%)	Samples
13	490.59	0.69	-234.72	1.33	41.20	12985
14	633.01	0.58	-179.92	1.29	43.70	12426
15	1059.82	0.41	-125.97	1.23	43.20	12242
16	1514.60	0.30	-137.83	1.29	45.65	12040

**Table 5.14:** Performance of strategy 13, 14, 15 and 16 for all samples in data set 1



**Figure 5.13:** Performance of strategy 13, 14, 15 and 16 for all samples in data set 1

Experiment 14 builds on experiment 13 and introduces all the available formula modifications that increased the remove rate during the lookup of *exact matches*, namely Canonization (Section 4.4), Variable Simplification (Section 4.6), Variable Propagation (Section 4.5) and Simplify replaced variables (Section 4.7.1).

The hit rate of experiment 14 did increase over the hit rate found in experiment 13 but the total time spent to do the extra modifications was not compensated for by the increase in performance of the SMT solver.

Experiment 15 then continues to build on experiment 14 with the added step of adding formulas to the cache immediately if they are not found (Section 3.9). A slight increase in remove rate was encountered but more interesting is the sharp increase in possible matches. This does imply that a lot of asserts from the same iteration share a very similar shape and only differs in variables. This coincidence in shape did not also reflect a coincidence in the shape of the assignments making up the rest of the formula.

Experiment 16 (Section A.16) attempts to make the assert less unique and move more of the terms that make up part of the assert into the assignments (Section 4.8). With a less unique assert the number of possible matches should increase and with more terms in the assignments, it should allow for smaller formulas to have a greater chance of being matched with the current formula. Experiment 16 did increase the number of possible matches nearly 5 times and obtained a greater remove rate. When examining the total time it is apparent that all these extra matches greatly increased the amount of time it would take to prove a subset match does or does not exist in the cache.

### Insight

An increase in remove rate and a decrease in filter steps do show the potential of all this extra work, but the extra cache hits did come at a greater cost to do all the modifications first.

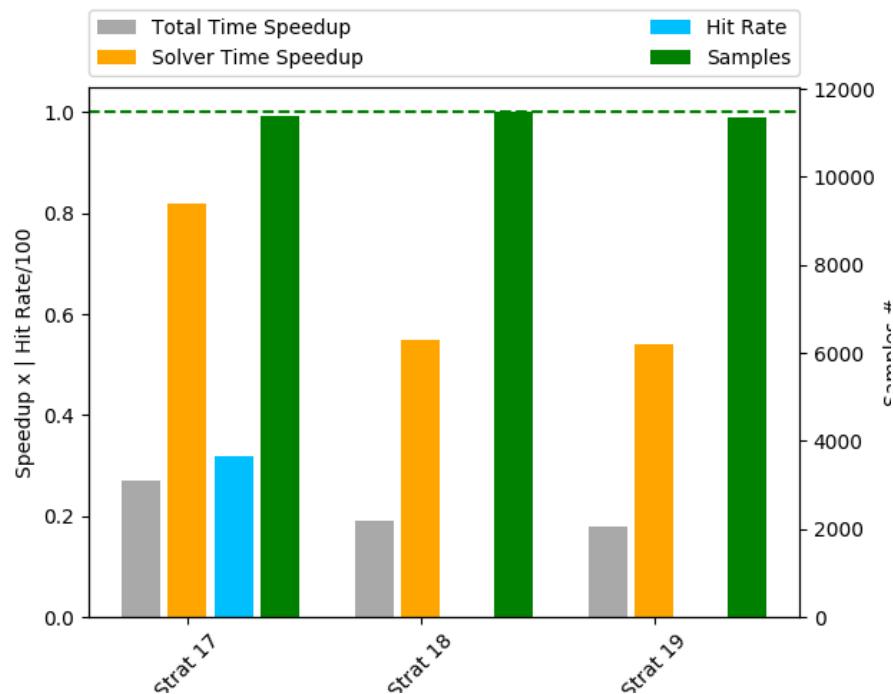
Interestingly, when looking at an assert in isolation, it would appear that most asserts from a single iteration share the exact same shape and only differ in variable names. A slight increase in cache hits did come at a greater cost in overall time though for which the SMT solver was not able to compensate.

### 5.4.9 Unsat Core Lookup

Detailed data can be found for strategy 17 on page 170, strategy 18 on page 173 and strategy 19 on page 176.

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
17	1831.98	0.27	123.09	0.82	31.97	11370
18	2117.02	0.19	354.51	0.55	0.07	11473
19	2051.20	0.18	358.42	0.54	0.00	11363

**Table 5.15:** Performance of strategy 17, 18 and 19 for all samples in data set 1



**Figure 5.14:** Performance of strategy 17, 18 and 19 for all samples in data set 1

Experiment 17 introduces the retrieval of unsat cores for the first time. Since it also built on subset matching, the first experiment will already enable all the filters as they have shown to be successful in previous experiments. The hit rate achieved by strategy 17 is performing quite well but there is no increase in solver speedup. As explained in Section 3.8, the fact that unsat-cores require labels to be added to the solver to track the related terms is causing the SMT solver to take longer to return a result as well each assert causing a separate solver call to enable the collection of individual unsat-cores. Experiment 18 and experiment 19 followed similar paths as previous experiments where the formula would be modified further to help increase

the reuse potential of the cache. Storing unsatisfiable cores behave slightly differently though, the unsatisfiable cores refer back to assignments and asserts that were added to the solver. These assignments and asserts are not modified to try and mitigate the unpredictability of the solver with regards to solving time.

### **Insight**

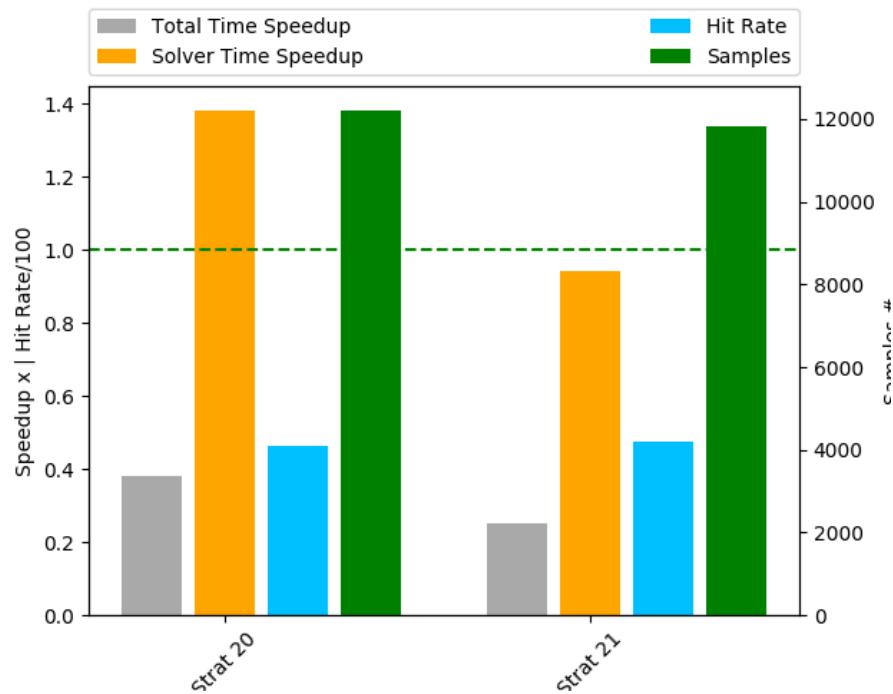
Unsat cores offer more reuse potential with a large number of possible matches, but calling the SMT solver multiple times makes it a losing battle, as no amount of performance increase in the cache should ever yield an overall performance increase above those encountered by the other strategies that have the same remove rate and a single solver call. (Section 2.5.3). Thus the formula being stored completely ignores the formula that is being modified in subsequent lookups. Any alterations to the formula during Unsat Lookups would greatly impact the ability to find a cache hit during the lookup process.

### 5.4.10 Hybrid Strategies

Detailed data can be found for strategy 20 on page 179 and strategy 21 on page 182.

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
20	1185.88	0.38	-189.12	1.38	46.51	12198
21	1690.64	0.25	31.36	0.94	47.45	11820

**Table 5.16:** Performance of strategy 20 and 21 for all samples in data set 1



**Figure 5.15:** Performance of strategy 20 and 21 for all samples in data set 1

Experiment 20 combines the options from experiment 4 and experiment 16 while experiment 21 also combines them but adds the options from experiment 19 as well. Experiment 21 now greatly benefits from the normal lookup and subset lookup but still loses time overall since it is still adding labels to track the unsatisfiable core.

**Insight**

Combining the strategies to look for exact matches and to look for partial matches gives the best of both worlds. The exact matches get rid of the most formulas and the ones that are left still get removed by the subset matching. The difference between the hybrid strategy and the normal lookup strategy might not make the hybrid strategy worth the extra time, although increases in performance might eventually change that as more cache hits on average should yield faster solving times from the SMT solvers.

### 5.4.11 Identifying significant overheads for Strategy 20 and 22

Strat	$\Delta TT$ Avg(ms)	$\Delta ST$ Avg(ms)	HR(%)	CT Avg(ms)	LT Avg(ms)	VRT Avg(ms)	Samples
Baseline solver time 0ms to 10ms							
20	594.78	-0.42	46.95	482.42	160.80	0.55	5861
22	409.62	-0.34	37.16	8.84	2.95	12.02	5955
Baseline solver time 11ms to 100ms							
20	818.79	-7.75	31.44	1826.27	608.76	2.29	2904
22	134.92	-4.40	25.05	20.25	6.75	60.35	3008
Baseline solver time 101ms to 1000ms							
20	2294.66	-93.28	58.62	4086.51	1362.17	3.90	2065
22	292.67	-65.63	52.57	62.341	20.78	159.09	2221
Baseline solver time 1s to 10s							
20	3495.52	-820.83	59.39	6305.60	2101.87	8.92	1219
22	-5.43	-888.97	58.81	274.62	91.54	378.41	1653
Baseline solver time 10s to 100s							
20	-2742.86	-7391.95	51.19	9361.81	3120.60	4.02	147
22	-2617.89	-4041.97	40.35	129.22	43.07	778.14	240
Baseline solver time 100s to 1000s							
20	2659.00	-1023.00	0.50	150.00	50.00	7.00	2
22	-28276.60	-30645.40	3.00	92.90	30.97	1613.80	5
Baseline solver time 0s to 1000s							
20	1185.89	-189.12	46.51	2147.52	715.84	2.46	12198
22	207.65	-210.50	39.84	57.63	19.20	111.64	13082

**Table 5.17:** Identifying overhead of strategy 20 and 22 for data set 1

Table 5.17 analyses the performance of strategy 20 and strategy 22 in more detail to identify the areas in which the cache is performing poorly so that it may be optimized in the future if possible. The main columns of interest are *CT Avg*, *LT Avg* and *VRT Avg*, each displaying the average amount of time it took per iteration to store formulas (CT Avg), lookup formulas (LT) and rename formulas (VRT).

Looking first at VRT Avg it is clear that experiment 22 takes more time on average. This is because strategy 20 only does variable renaming for the key before it is stored while strategy 22 is renaming the entire formula. Of the 207ms that strategy 22 is slower on average per iteration 111ms can be attributed just to variable renaming. It is not immediately clear if there is a much better way of decreasing the number of time it takes to rename a formula but the bulk of the time is spent here. For strategy 20 it is not worth going into detail with regards to improving the renaming process but any gains made for strategy 22 in this regards will also slightly improve this part of strategy 20.

Next looking at LT Avg strategy 22 is not losing too much time per iteration but strategy 20 is loosing a lot of time. The lookup time for strategy 20 includes the process of subset matching over all possible formulas to see if a formula is stored that is considered a subset match. As expected this is taking up a large amount of time for strategy 20 and can perhaps be improved on further in the future by decreasing the amount of samples to compare to and to introduce more sophisticated filters. Strategy 22, however, has a much smaller time that is required to look it up but still not constant as would be preferred, since it doesn't include the time take to do the variable

renaming. During the analysis of the experiment data, there were 0 collisions detected for the keys in strategy 22, this indicates that the current method of mapping formulas where the crc value is used and then the full formula can be replaced with just the crc value for reuse within a single run, once the cache is made persistent this will have to be evaluated again.

Analyzing CT Avg does show some surprising results as storing the formulas is taking longer than looking them up. It is possible that some work is timed twice as part of the storing process also looks to see if the key already exists but even then it is still slightly high for strategy 22, replacing it with the crc value should decrease the amount of time taken here as well bringing strategy 22 to an almost net equal over all results while giving further improvements over the time ranges where it is already giving a performance increase. Strategy 20, on the other hand, is dominated by the time taken to store a formula. This points to an efficiency problem in the code base as the amount of time it takes to store a formula should not be so lopsided. Currently, formulas for each key are stored in a vector and with the vector growing often it could be that there is a lot of memory being allocated and reallocated to accommodate for the constant increase of formulas. This should be the starting point of any performance enhancements.

Curiously by summing over the total time difference, solver time and the amount of time all the operations take it does not equal to 0. The solver is not making up the time difference that is appearing in the data. A procedure that was no longer timed for these operations was the time it took to encode the formulas to the solver. It would appear that the smaller formulas are faster to encode and is giving an extra speedup that was not expected. This will need to be confirmed with more detailed experiments.

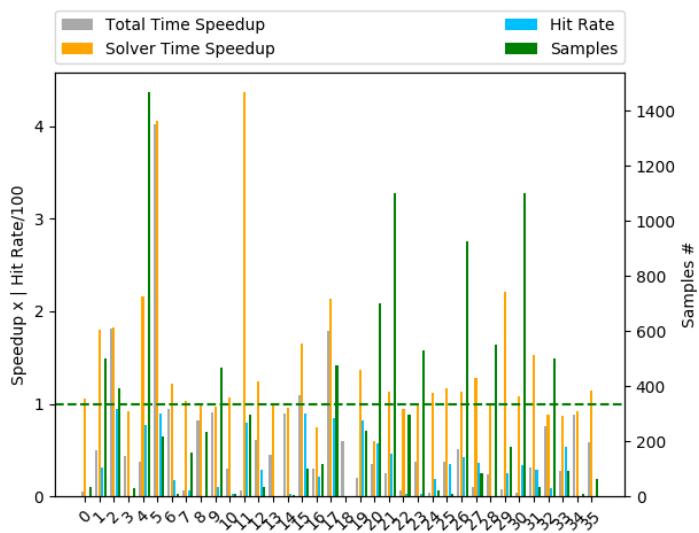
### 5.4.12 Identifying categories where caching works better

Label	Folder	TT x	ST x	HR	Samples
0	ssh	0.06	1.06	0.00	36
1	loop-invgen	0.50	1.80	30.77	500
2	reducercommutativity	1.81	1.83	94.34	393
3	termination-libowfat	0.44	0.92	0.23	31
4	seq-mthreaded	0.37	2.17	77.51	1467
5	floats-cdfpl	4.02	4.05	89.91	218
6	termination-crafted-lit	0.95	1.22	17.00	9
7	systemc	0.07	1.04	6.45	161
8	loop-new	0.82	0.98	0.00	236
9	loop-lit	0.91	0.96	10.45	469
10	memsafety-ext	0.30	1.07	2.88	8
11	ntdrivers-simplified	0.07	4.37	80.00	295
12	seq-pthread	0.61	1.25	28.36	36
13	memory-alloca	0.44	1.00	0.00	1
14	ntdrivers	0.90	0.95	2.75	4
15	termination-crafted	1.09	1.65	89.32	99
16	array-memsafety	0.30	0.75	21.54	118
17	float-benchs	1.78	2.13	84.00	478
18	ldv-challenges	0.60	0.00	0.00	1
19	memsafety	0.20	1.36	81.52	238
20	array-examples	0.35	0.59	57.27	702
21	loops	0.25	1.13	45.62	1103
22	recursive	0.06	0.95	2.19	297
23	ssh-simplified	0.37	1.01	2.55	529
24	ldv-commit-tester	0.04	1.12	18.35	23
25	pthread	0.38	1.16	35.27	11
26	bitvector	0.51	1.13	42.45	925
27	ldv-consumption	0.11	1.28	36.47	83
28	locks	0.24	1.00	0.00	552
29	termination-numeric	0.08	2.21	24.87	182
30	recursive-simple	0.04	1.08	33.51	1101
31	heap-manipulation	0.31	1.52	29.36	36
32	loop-acceleration	0.76	0.88	8.37	499
33	bitvector-regression	0.28	0.87	53.85	91
34	bitvector-loops	0.89	0.92	0.00	11
35	termination-15	0.59	1.15	0.81	62

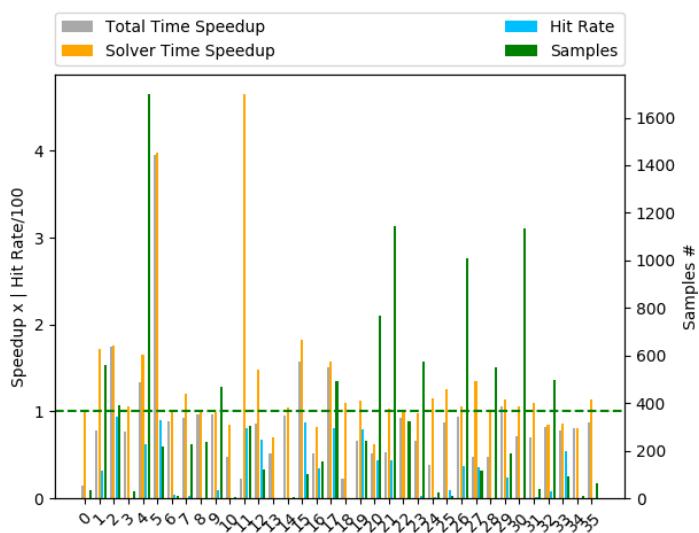
**Table 5.18:** Performance of strategy 20 across all categories for all samples in data set 1

Label	Folder	TT x	ST x	HR	Samples
0	ssh	0.15	0.99	0.00	36
1	loop-invgen	0.78	1.72	31.71	560
2	reducercommutativity	1.75	1.76	93.95	393
3	termination-libowfat	0.77	1.06	0.00	31
4	seq-mthreaded	1.34	1.65	62.93	1698
5	floats-cdfpl	3.95	3.98	89.91	218
6	termination-crafted-lit	0.88	1.00	3.67	9
7	systemc	0.93	1.20	2.41	229
8	loop-new	0.96	1.01	0.00	238
9	loop-lit	0.97	0.99	9.65	469
10	memsafety-ext	0.48	0.84	0.00	8
11	ntdrivers-simplified	0.23	4.65	80.59	304
12	seq-pthread	0.86	1.49	67.83	121
13	memory-alloca	0.52	0.70	0.00	1
14	ntdrivers	0.95	1.05	0.00	6
15	termination-crafted	1.57	1.82	88.06	104
16	array-memsafety	0.51	0.82	34.69	157
17	float-benchs	1.50	1.58	80.28	492
18	ldv-challenges	0.22	1.09	0.00	2
19	memsafety	0.66	1.12	80.09	240
20	array-examples	0.52	0.63	43.66	770
21	loops	0.53	1.03	43.22	1146
22	recursive	0.93	1.02	0.15	326
23	ssh-simplified	0.66	0.98	3.41	573
24	ldv-commit-tester	0.39	1.15	1.00	23
25	pthread	0.88	1.25	9.09	11
26	bitvector	0.94	1.06	37.10	1008
27	ldv-consumption	0.47	1.35	35.32	116
28	locks	0.48	1.02	0.00	552
29	termination-numeric	1.06	1.13	24.07	188
30	recursive-simple	0.71	1.06	0.09	1134
31	heap-manipulation	0.70	1.10	2.08	40
32	loop-acceleration	0.82	0.85	8.37	499
33	bitvector-regression	0.79	0.87	53.85	91
34	bitvector-loops	0.80	0.81	0.00	11
35	termination-15	0.88	1.14	0.00	62

**Table 5.19:** Performance of strategy 22 across all categories for all samples in data set 1



**Figure 5.16:** Performance of strategy 20 across all categories for all samples in data set 1



**Figure 5.17:** Performance of strategy 22 across all categories for all samples in data set 1

Label	Folder	TT x	ST x	HR	Samples
2	reducercommutativity	1.81	1.83	94.34	393
4	seq-mthreaded	0.37	2.17	77.51	1467
5	floats-cdfpl	4.02	4.05	89.91	218
11	ntdrivers-simplified	0.07	4.37	80.00	295
15	termination-crafted	1.09	1.65	89.32	99
17	float-benchs	1.78	2.13	84.00	478
29	termination-numeric	0.08	2.21	24.87	182

**Table 5.20:** Performance of strategy 20 across good categories for all samples in data set 1

Label	Folder	TT x	ST x	HR	Samples
2	reducercommutativity	1.75	1.76	93.95	393
4	seq-mthreaded	1.34	1.65	62.93	1698
5	floats-cdfpl	3.95	3.98	89.91	218
11	ntdrivers-simplified	0.23	4.65	80.59	304
15	termination-crafted	1.57	1.82	88.06	104
17	float-benchs	1.50	1.58	80.28	492
29	termination-numeric	1.06	1.13	24.07	188

**Table 5.21:** Performance of strategy 22 across good categories for all samples in data set 1

For the most part, it would appear that the cache performs very well when the hit rate is getting close to a 100 hit rate (even though a 100 is not possible). This is an indication that the formulas in these categories are the same formulas during subsequent iterations, allowing for the cache to eliminate the need for the solver call entirely. These categories are mostly centered around floats and samples where there is a specific point of termination. Floats are known to be extremely computationally expensive even for small formulas, it can clearly be observed here as well that the TT speedup and ST speedup time are very consistent for strategy 20 and 22 implying the formulas are not very large and do not suffer a very large overhead cost. An interesting difference between the two strategies appeared in the *termination-numeric* category, the large difference in TT x could be an indication that there were a lot of formulas to lookup and store and thus strategy 20 suffered from more overhead in total. Some of the time lost was made up for by the solver (implying again that the solver was not needed for some samples).

## 5.5 Data Set 2

Data set 2 is an extended version of data set 1 (Section 5.4) but with the results from concurrency added as well.

### 5.5.1 Setup

#### SMT Solver: Z3

For data set 2 an Intel 5820k 6 core multithreaded CPU at 3.3 GHz with 16GB of RAM will be used to perform all experiments on. Each run was limited to 3 minutes and with a 2GB memory cap to get initial estimates before larger experiments were run on a cluster. All verification runs were made across the SV-COMP 2016 data set with the following parameters along with the parameter specified in the setup of the experiment in their relevant sections. Each verification run starts with an empty cache. For data set 1 only the samples containing concurrency data has been included. The SMT solver used during this data set is Z3.

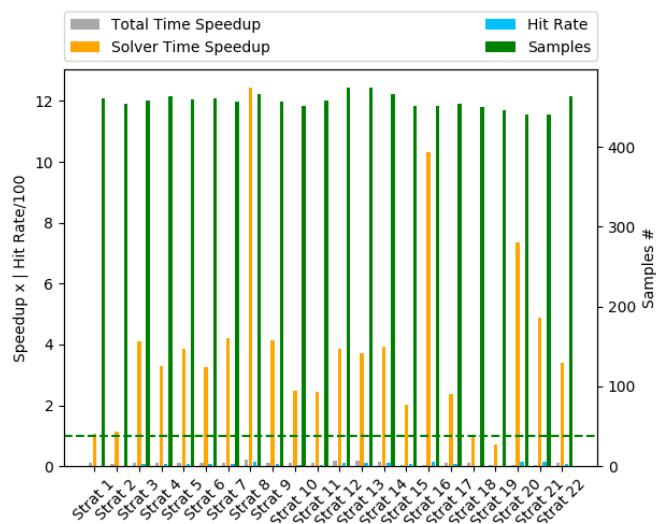
```
--bv --z3 --falsification --timeout 180 --memlimit 2g
```

### 5.5.2 Results

#### Experiments where the baseline had a solving time between 0s and 1000s

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	3330.48	0.10	-2.19	1.08	0.00	461
2	3687.80	0.08	-3.46	1.14	0.00	454
3	4250.88	0.13	-35.00	4.11	7.84	458
4	4531.80	0.13	-36.98	3.30	8.47	463
5	4524.60	0.12	-33.50	3.86	7.85	459
6	4460.09	0.13	-35.92	3.25	8.14	461
7	4556.84	0.12	-33.70	4.21	8.03	457
8	4276.20	0.21	-54.65	12.43	15.87	466
9	4531.73	0.12	-33.55	4.15	8.03	457
10	4078.94	0.12	-29.17	2.47	2.53	452
11	4307.78	0.12	-29.53	2.46	3.12	458
12	4391.61	0.18	-43.83	3.84	10.45	474
13	4379.60	0.18	-43.26	3.71	10.52	474
14	4400.94	0.15	-40.79	3.92	9.78	466
15	4542.09	0.06	-13.36	2.01	7.21	452
16	4836.16	0.06	-18.81	10.33	15.15	451
17	6366.18	0.10	-28.70	2.37	6.53	454
18	3812.01	0.10	-0.26	1.01	1.06	450
19	4171.77	0.06	5.80	0.73	0.00	446
20	4307.69	0.04	-11.04	7.35	13.15	440
21	4507.78	0.04	-10.16	4.87	13.15	440
22	4788.21	0.13	-37.43	3.39	8.47	463

**Table 5.22:** Performance of all strategies where the baseline had a solving time between 0s and 1000s for data set 2



**Figure 5.18:** Performance of all strategies where the baseline had a solving time between 0s and 1000s for data set 2

### Experiments where the baseline had a solving time between 0ms and 10ms

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	1.21	0.80	0.03	0.88	0.00	390
2	1.80	0.73	0.01	0.97	0.00	390
3	2.78	0.63	-0.07	1.47	2.00	389
4	2.39	0.67	-0.06	1.38	2.05	390
5	4.33	0.52	-0.06	1.38	2.03	390
6	5.11	0.48	-0.06	1.38	2.05	390
7	5.85	0.45	-0.06	1.40	2.05	390
8	65.36	0.06	-0.11	2.56	3.66	387
9	5.59	0.46	-0.07	1.45	2.05	390
10	35.88	0.12	0.02	0.90	0.49	390
11	12.16	0.28	-0.05	1.31	0.62	390
12	2.67	0.64	-0.06	1.35	2.08	390
13	2.25	0.68	-0.06	1.38	2.08	390
14	6.61	0.42	-0.05	1.31	2.08	390
15	7.17	0.40	-0.08	1.62	2.08	390
16	8.47	0.36	-0.09	1.79	3.81	390
17	6.52	0.42	0.03	0.87	1.33	390
18	5.72	0.46	0.04	0.85	0.46	390
19	6.40	0.43	0.13	0.63	0.00	390
20	10.57	0.31	-0.10	1.91	3.81	390
21	11.30	0.30	-0.09	1.75	3.81	390
22	2.21	0.68	-0.05	1.27	2.05	390

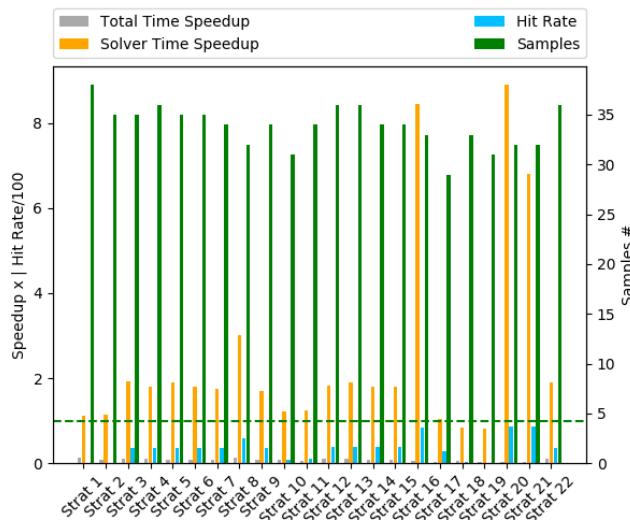
**Table 5.23:** Performance of all strategies where the baseline had a solving time between 0ms and 10ms for data set 2



**Figure 5.19:** Performance of all strategies where the baseline had a solving time between 0ms and 10ms for data set 2

**Experiments where the baseline had a solving time between 11ms and 100ms**

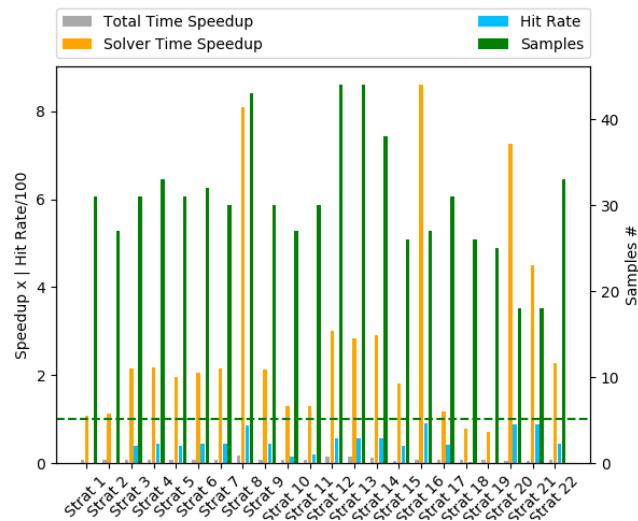
Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	8976.34	0.12	-5.16	1.12	0.00	38
2	12125.29	0.09	-6.34	1.15	0.00	35
3	13774.94	0.10	-24.37	1.93	36.26	35
4	13209.22	0.11	-22.53	1.81	36.75	36
5	19204.20	0.08	-23.97	1.90	36.51	35
6	13767.66	0.08	-21.80	1.79	35.31	35
7	15104.88	0.08	-21.24	1.75	36.38	34
8	9212.00	0.12	-33.16	3.02	58.56	32
9	15010.24	0.08	-20.29	1.69	36.38	34
10	16568.74	0.07	-8.74	1.21	8.26	31
11	17927.21	0.06	-9.88	1.25	9.68	34
12	14376.11	0.10	-22.78	1.82	37.33	36
13	14287.47	0.10	-23.78	1.89	37.78	36
14	12838.74	0.09	-22.00	1.80	38.62	34
15	17429.88	0.07	-21.91	1.79	38.62	34
16	24622.73	0.05	-44.09	8.46	85.12	33
17	41602.83	0.03	-2.52	1.05	28.38	29
18	23964.61	0.05	9.58	0.84	3.00	33
19	30317.42	0.04	11.26	0.81	0.00	31
20	35459.91	0.03	-44.91	8.90	85.38	32
21	36848.66	0.03	-43.16	6.80	85.38	32
22	13949.17	0.10	-24.06	1.91	36.75	36

**Table 5.24:** Performance of all strategies where the baseline had a solving time between 11ms and 100ms for data set 2**Figure 5.20:** Performance of all strategies where the baseline had a solving time between 11ms and 100ms for data set 2

### Experiments where the baseline had a solving time between 101ms and 1000ms

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	35196.77	0.08	-14.39	1.07	0.00	31
2	36812.74	0.08	-26.44	1.13	0.00	27
3	45270.94	0.08	-133.81	2.16	40.65	31
4	43829.79	0.09	-137.03	2.19	45.03	33
5	42943.94	0.08	-114.84	1.95	40.26	31
6	43073.75	0.09	-123.72	2.05	43.94	32
7	49596.80	0.08	-121.77	2.15	44.83	30
8	37388.05	0.18	-257.56	8.08	86.26	43
9	49134.10	0.08	-120.57	2.12	44.83	30
10	38524.74	0.07	-51.96	1.30	15.04	27
11	38900.60	0.08	-50.30	1.29	18.97	30
12	32449.30	0.16	-179.00	3.00	56.45	44
13	32390.55	0.16	-174.07	2.84	56.91	44
14	36614.82	0.13	-162.16	2.92	55.79	38
15	44444.08	0.06	-94.04	1.82	39.08	26
16	46316.70	0.07	-183.52	8.60	90.44	27
17	49035.55	0.09	-35.00	1.17	40.81	31
18	34974.46	0.09	60.08	0.78	3.81	26
19	36730.96	0.09	87.44	0.71	0.00	25
20	42030.11	0.05	-187.89	7.27	87.33	18
21	44436.56	0.05	-169.56	4.51	87.33	18
22	46242.88	0.08	-141.21	2.28	45.03	33

**Table 5.25:** Performance of all strategies where the baseline had a solving time between 101ms and 1000ms for data set 2

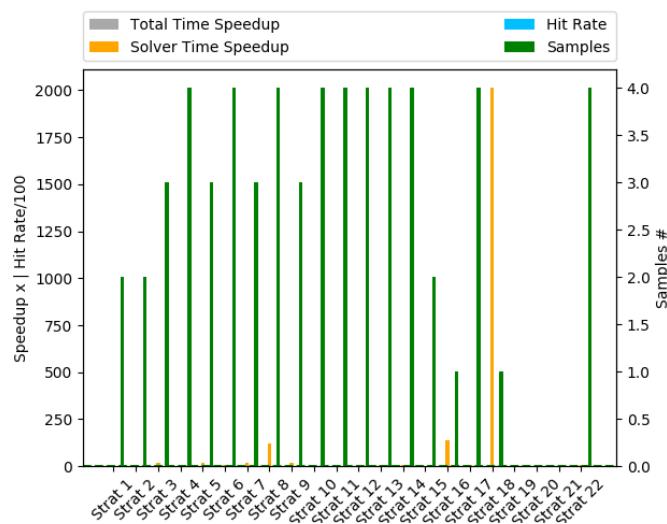


**Figure 5.21:** Performance of all strategies where the baseline had a solving time between 101ms and 1000ms for data set 2

### Experiments where the baseline had a solving time between 1s and 10s

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
1	51340.00	0.12	-188.50	1.08	0.00	2
2	127615.00	0.05	-320.00	1.15	0.00	2
3	20099.67	0.65	-3666.67	19.74	95.33	3
4	43843.25	0.40	-2942.00	5.53	78.00	4
5	23897.67	0.61	-3651.33	18.30	95.33	3
6	48471.00	0.38	-2953.00	5.62	78.00	4
7	26242.33	0.59	-3667.33	19.81	95.67	3
8	16236.00	0.68	-3321.75	119.63	99.00	4
9	28149.67	0.57	-3666.67	19.74	95.67	3
10	68972.00	0.30	-2880.00	5.05	72.25	4
11	47919.25	0.38	-2915.25	5.31	72.25	4
12	33818.50	0.47	-3014.75	6.23	78.50	4
13	33880.75	0.47	-2992.00	5.99	78.50	4
14	55095.50	0.35	-3019.75	6.28	78.50	4
15	151033.50	0.05	-1408.00	2.40	60.50	2
16	114705.00	0.06	-2037.00	136.80	99.00	1
17	40279.25	0.46	-2971.25	8.85	89.25	4
18	13007.00	0.85	-2012.00	2013.00	99.00	1
19	0.00	0.00	0.00	0.00	0.00	0
20	0.00	0.00	0.00	0.00	0.00	0
21	0.00	0.00	0.00	0.00	0.00	0
22	46973.00	0.39	-2946.75	5.57	78.00	4

**Table 5.26:** Performance of all strategies where the baseline had a solving time between 1s and 10s for data set 2



**Figure 5.22:** Performance of all strategies where the baseline had a solving time between 1s and 10s for data set 2

### 5.5.3 Insights

Verification-runs on multi-threaded programs have some interesting characteristics. During a single iteration, all possible interleavings are created and sent to the SMT solver. It can be expected for many of the same formulas to be produced during a single iteration. However, unlike the previous assumption, these formulas are no longer ordered as expected during subsequent unwindings. These formulas would perhaps perform better when a canonical form is applied to them. This is also reflected in the data as the hit rate was lower than that found during the analysis of data set 1. Another interesting result that came to the fore is the solver speedup being substantially higher on average compared to that seen in data set 1. This can be attributed to the fact that there may have been few cache hits, but that each hit removed all terms from the disjunct of asserts, causing no solver call and giving a net gain in terms of solver performance.

The total time does show the negative impact of having all the extra overhead during a single iteration with multiple formulas and solver calls. There were no results present for any samples where the solver took more than 10s and very few samples where the solver took between 1s and 10s so it is unclear whether it would follow a similar trend as to data set 1 where substantially harder problems don't just gain performance in terms of solver time but also started to give a net performance increase for the total amount of time spent by *EBMC*.

During data set 1 a surprisingly good performer was strategy 1 that took the entire formula as is and cached it. Although it did achieve limited success with the number of cache hits it would register, it was very effective in terms of the amount of time it took to store and retrieve the formulas, where any hit resulted in the solver not having to be called. It is interesting that neither strategy 1 nor strategy 2 got any cache hits at all, reaffirming the previous point that canonization is most likely needed for these problems as no formulas appear to be the same in subsequent iterations.

## 5.6 Data Set 3

### 5.6.1 Setup

#### SMT Solver: Z3

For data set 3, all experiments were run on the IRIDIS 4 cluster at the University of Southampton. The experiments were distributed across computing nodes which each contained 16 CPUs clocked at 2.6GHz with a total of 64GB of memory. Each run was limited to 15 minutes with a 15GB memory cap with only the baseline, strategy 20 and strategy 22 being computed. All verification runs were made across the SV-COMP 2017 data set with the following parameters along with the parameter specified in the setup of the experiment in their relevant sections. Each verification run starts with an empty cache. The SMT solver used during this data set is Z3.

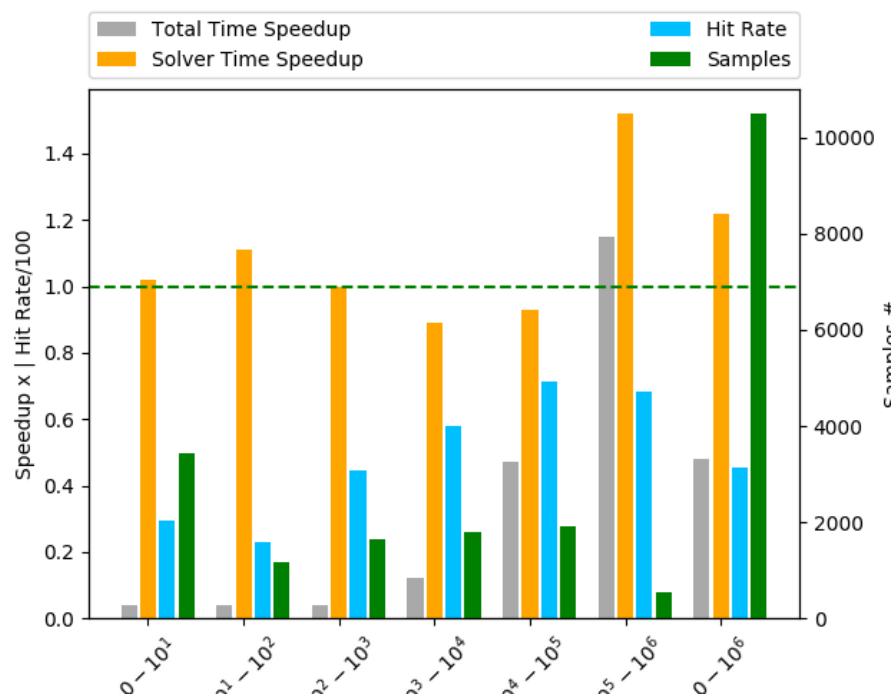
```
--no-div-by-zero-check --timeout 895s --memlimit 15g --no-align-check
--force-malloc-success --unlimited-k-steps --state-hashing --floatbv
--bv --z3 --falsification --cache
```

### 5.6.2 Results

Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
Baseline solver time 0ms to 10ms						
20	3148.85	0.04	-0.05	1.02	29.35	3425
22	767.67	0.15	-0.14	1.06	23.82	3378
Baseline solver time 11ms to 100ms						
20	19359.15	0.04	-4.11	1.11	23.01	1175
22	14652.68	0.15	-1.72	1.04	12.59	1258
Baseline solver time 101ms to 1000ms						
20	34322.60	0.04	-1.30	1.00	44.68	1650
22	12159.51	0.19	-8.23	1.02	17.87	1620
Baseline solver time 1s to 10s						
20	42273.07	0.12	467.06	0.89	57.96	1782
22	18722.03	0.29	697.40	0.85	35.39	1872
Baseline solver time 10s to 100s						
20	43146.51	0.47	2766.68	0.93	71.20	1924
22	31166.15	0.56	8572.70	0.80	51.93	1975
Baseline solver time 100s to 1000s						
20	-30245.31	1.15	-78383.60	1.52	68.49	538
22	-3557.95	1.02	-34054.53	1.18	49.63	594
Baseline solver time 0s to 1000s						
20	22130.42	0.48	-3432.64	1.22	45.59	10494
22	12640.18	0.64	-187.69	1.01	30.25	10697

**Table 5.27:** Performance of all strategies over all samples in data set 3

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	3148.85	0.04	-0.05	1.02	29.35	3425
$10^1 - 10^2$	19359.15	0.04	-4.11	1.11	23.01	1175
$10^2 - 10^3$	34322.60	0.04	-1.30	1.00	44.68	1650
$10^3 - 10^4$	42273.07	0.12	467.06	0.89	57.96	1782
$10^4 - 10^5$	43146.51	0.47	2766.68	0.93	71.20	1924
$10^5 - 10^6$	-30245.31	1.15	-78383.60	1.52	68.49	538
$0 - 10^6$	22130.42	0.48	-3432.64	1.22	45.59	10494

**Table 5.28:** Performance of strategy 20 over all samples in data set 3**Figure 5.23:** Performance of strategy 20 over all samples in data set 3

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	767.67	0.15	-0.14	1.06	23.82	3378
$10^1 - 10^2$	14652.68	0.15	-1.72	1.04	12.59	1258
$10^2 - 10^3$	12159.51	0.19	-8.23	1.02	17.87	1620
$10^3 - 10^4$	18722.03	0.29	697.40	0.85	35.39	1872
$10^4 - 10^5$	31166.15	0.56	8572.70	0.80	51.93	1975
$10^5 - 10^6$	-3557.95	1.02	-34054.53	1.18	49.63	594
$0 - 10^6$	12640.18	0.64	-187.69	1.01	30.25	10697

**Table 5.29:** Performance of strategy 22 over all samples in data set 3

## 5.6. DATA SET 3

95

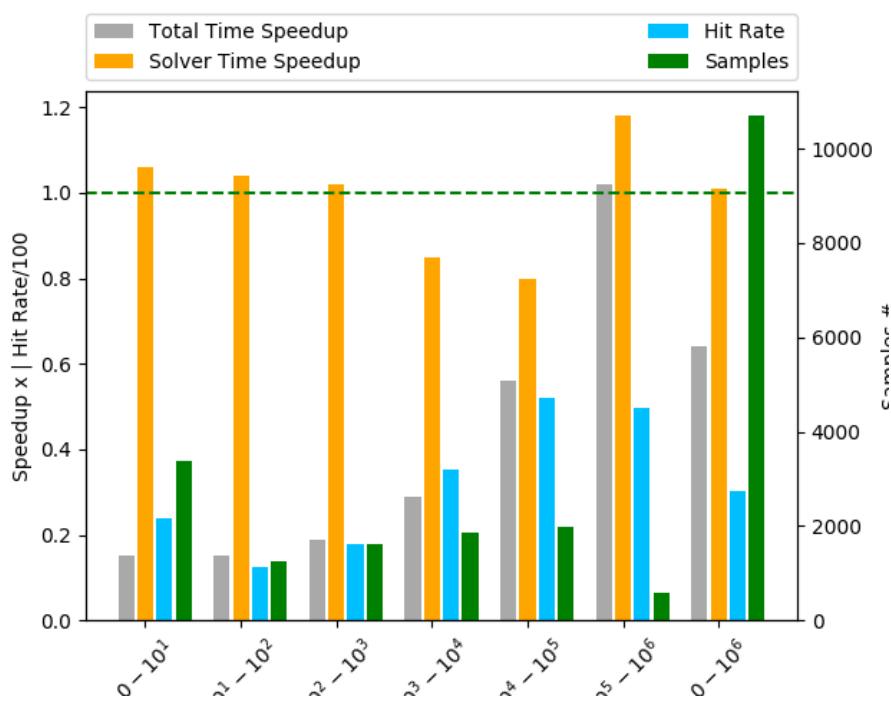


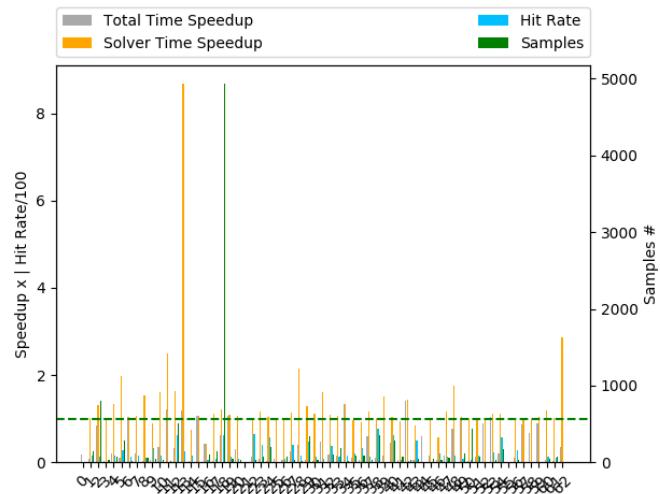
Figure 5.24: Performance of strategy 22 over all samples in data set 3

Label	Folder	TT x	ST x	HR	Samples
0	psyco	0.18	0.00	0.00	1
1	forester-heap	0.07	1.02	15.41	141
2	ldv-linux-3.4-simple	0.85	1.30	12.50	806
3	ssh	0.02	1.00	0.00	36
4	loop-invgem	0.20	1.34	15.21	68
5	ldv-linux-3.14	0.10	1.97	28.81	281
6	reducercommutativity	1.03	1.04	13.69	13
7	ldv-linux-3.7.3	0.20	1.06	14.40	5
8	termination-libowfat	0.03	1.52	10.92	63
9	ldv-regression	0.05	0.89	32.49	49
10	ldv-linux-3.0	0.36	1.62	15.22	27
11	signedintegeroverflow-regression	1.22	2.50	0.00	10
12	seq-mthreaded	0.31	1.64	61.76	507
13	floats-esbmc-regression	1.20	8.68	26.00	3
14	ldv-validator-v0.8	0.03	0.75	14.40	5
15	floats-cdfpl	1.06	1.06	0.00	6
16	termination-crafted-lit	0.43	0.43	4.30	101
17	systemc	0.04	1.11	7.16	139
18	eca-rers2012	0.62	1.21	62.44	4934
19	loop-lit	1.05	1.08	13.43	40
20	memsafety-ext	0.30	1.07	6.89	35
21	array-industry-pattern	0.00	0.00	0.00	0
22	ldv-memsafety-bitfields	0.13	1.00	65.19	32
23	ldv-linux-3.12-rc1	0.07	1.16	40.08	78
24	ldv-linux-4.2-rc1	0.03	1.05	57.72	193
25	ntdrivers-simplified	0.09	1.02	0.00	4
26	list-ext-properties	0.06	1.00	8.33	70
27	list-ext2-properties	0.25	1.15	39.04	25
28	seq-pthread	0.40	2.14	15.77	22
29	ldv-linux-3.16-rc1	0.06	1.28	46.91	343
30	termination-memory-alloc	0.04	1.12	11.83	35
31	ntdrivers	0.46	1.62	7.80	5
32	termination-crafted	0.17	1.09	38.07	98
33	array-memsafety	0.15	1.07	14.03	191
34	float-benchs	1.33	1.33	15.32	19
35	ldv-challenges	0.11	0.96	20.22	91
36	memsafety	0.06	0.93	31.97	89
37	ldv-linux-4.0-rc1-mav	0.60	1.16	12.93	29
38	busybox-1.22.0	0.11	1.02	75.70	360
39	array-examples	0.14	1.50	0.00	2
40	ldv-memsafety	0.44	1.04	61.13	285
41	loops	0.06	0.95	4.62	68
42	recursive	1.42	1.44	2.08	24
43	ldv-validator-v0.6	0.05	0.85	48.65	43
44	ddv-machzwd	0.60	0.00	0.00	3
45	ssh-simplified	0.15	1.00	0.00	43
46	ldv-commit-tester	0.05	0.58	19.42	24
47	memsafety-ext2	0.16	1.16	14.04	52
48	pthread	0.76	1.77	14.33	3
49	bitvector	1.01	1.03	7.90	115
50	product-lines	0.02	1.02	6.53	437
51	ldv-consumption	0.14	0.99	16.40	75
52	loop-industry-pattern	0.89	1.01	0.00	6
53	termination-numeric	1.05	1.12	23.11	28
54	recursive-simple	0.19	1.12	58.18	165
55	termination-memory-linkedlists	0.00	0.00	0.00	0
56	heap-manipulation	0.09	0.98	27.46	26
57	loop-acceleration	0.88	1.00	0.00	6
58	bitvector-regression	0.67	1.00	0.00	5
59	bitvector-loops	0.88	1.04	0.00	2
60	list-properties	0.05	1.20	13.85	46
61	termination-recursive-malloc	0.03	1.02	10.95	75
62	termination-15	0.34	2.86	0.00	7

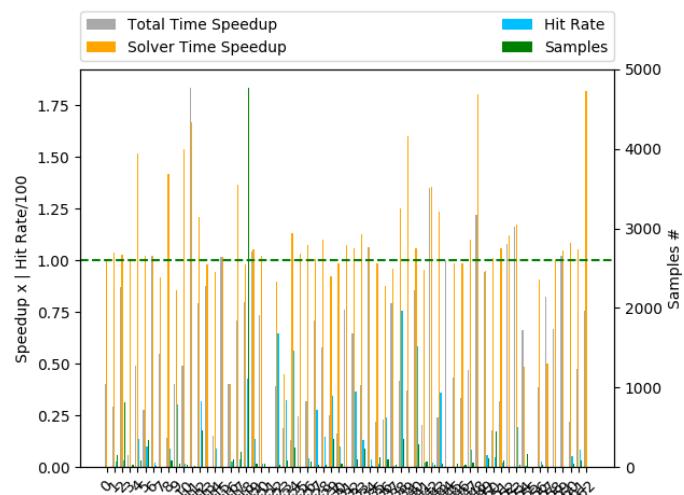
Table 5.30: Performance of strategy 20 across all categories for all samples in data set 3

Label	Folder	TT x	ST x	HR	Samples
0	psyco	0.40	1.00	0.00	1
1	forester-heap	0.29	1.04	2.79	149
2	ldv-linux-3.4-simple	0.87	1.03	3.03	813
3	ssh	0.06	1.00	0.00	36
4	loop-invgen	0.49	1.51	13.75	81
5	ldv-linux-3.14	0.28	1.02	9.96	347
6	reducercommutativity	1.02	1.02	2.00	13
7	ldv-linux-3.7.3	0.55	0.92	0.33	6
8	termination-libowfat	0.14	1.42	9.17	87
9	ldv-regression	0.40	0.85	30.49	49
10	ldv-linux-3.0	0.49	1.54	1.76	29
11	signedintegeroverflow-regression	1.83	1.67	0.00	10
12	seq-mthreaded	0.79	1.21	31.82	465
13	floats-esbmc-regression	0.87	0.98	0.00	3
14	ldv-validator-v0.8	0.15	0.94	8.86	7
15	floats-cdfpl	1.02	1.02	0.00	6
16	termination-crafted-lit	0.40	0.40	2.68	101
17	systemc	0.71	1.37	3.75	198
18	eca-rers2012	0.80	0.98	42.85	4766
19	loop-lit	1.04	1.05	13.43	40
20	memsafety-ext	0.74	1.02	1.45	38
21	array-industry-pattern	0.00	0.00	0.00	0
22	ldv-memsafety-bitfields	0.39	0.89	64.47	32
23	ldv-linux-3.12-rc1	0.19	0.45	32.16	79
24	ldv-linux-4.2-rc1	0.13	1.13	56.32	251
25	ntdrivers-simplified	0.24	1.03	0.00	4
26	list-ext-properties	0.32	1.08	4.28	74
27	list-ext2-properties	0.71	1.00	27.62	26
28	seq-pthread	0.58	1.10	14.85	34
29	ldv-linux-3.16-rc1	0.25	0.92	34.54	352
30	termination-memory-alloca	0.16	0.99	10.13	46
31	ntdrivers	0.76	1.07	0.00	5
32	termination-crafted	0.65	1.06	36.45	98
33	array-memsafety	0.40	1.13	13.09	238
34	float-benchs	1.06	1.06	3.95	21
35	ldv-challenges	0.22	0.98	1.62	124
36	memsafety	0.23	0.88	24.24	91
37	ldv-linux-4.0-rc1-mav	0.79	0.96	0.59	32
38	busybox-1.22.0	0.42	1.25	75.70	360
39	array-examples	0.37	1.60	0.00	2
40	ldv-memsafety	0.85	1.06	58.61	285
41	loops	0.20	0.96	2.33	72
42	recursive	1.35	1.36	2.08	24
43	ldv-validator-v0.6	0.24	1.23	36.13	45
44	ddv-machzwd	1.00	0.00	0.00	3
45	ssh-simplified	0.43	0.99	0.00	43
46	ldv-commit-tester	0.34	0.99	0.88	26
47	memsafety-ext2	0.47	1.10	8.56	52
48	pthread	1.22	1.80	0.00	3
49	bitvector	0.94	0.95	5.76	114
50	product-lines	0.18	1.01	4.87	446
51	ldv-consumption	0.32	1.06	2.19	88
52	loop-industry-pattern	1.08	1.12	0.00	6
53	termination-numeric	1.16	1.17	19.43	28
54	recursive-simple	0.66	0.49	0.61	165
55	termination-memory-linkedlists	0.00	0.00	0.00	0
56	heap-manipulation	0.39	0.91	2.67	30
57	loop-acceleration	0.82	0.50	0.00	6
58	bitvector-regression	0.67	1.00	0.00	5
59	bitvector-loops	1.02	1.05	0.00	2
60	list-properties	0.22	1.09	5.12	48
61	termination-recursive-malloc	0.47	1.05	8.19	85
62	termination-15	0.75	1.82	0.00	7

**Table 5.31:** Performance of strategy 22 across all categories for all samples in data set 3



**Figure 5.25:** Performance of strategy 20 across all categories for all samples in data set 3



**Figure 5.26:** Performance of strategy 22 across all categories for all samples in data set 3

### 5.6.3 Insight

The first thing that is noticed when evaluating the performance of strategy 20 and strategy 22 with regards to their performance in data set 3 is that they did considerably worse than they did in data set 1. The solverZ3 still fluctuated quite a bit as was expected from the initial data set but the categories that clearly performed well in data set 1 was noticeably absent from any promising results in data set 2. The only difference between the data sets that could have caused this significant variation in performance time was the addition of the `--floatbv` flag. Without this flag, floats were encoded as fixed-point numbers, which may be faster but are only approximations. This has caused a lot of experiments to time out on the cluster and an accurate conclusion cannot be made at this time until those experiments are run with longer timeouts. Overall the cache still showed that it is effective when it comes to matching formulas that are already in the cache but the time gained from these matches only pay off for the harder problems.

## 5.7 Data Set 4

### 5.7.1 Setup

#### SMT Solver: boolector

For data set 4, all experiments were run on the IRIDIS 4 cluster at the University of Southampton. The experiments were distributed across computing nodes which each contained 16 CPUs clocked at 2.6GHz with a total of 64GB of memory. Each run was limited to 15 minutes with a 15GB memory cap with only the baseline, strategy 20 and strategy 22 being computed. All verification runs were made across the SV-COMP 2017 data set with the following parameters along with the parameter specified in the setup of the experiment in their relevant sections. Each verification run starts with an empty cache. The SMT solver used during this data set is boolector.

```
--no-div-by-zero-check --timeout 895s --memlimit 15g --no-align-check
--force-malloc-success --unlimited-k-steps --state-hashing --floatbv
--bv --z3 --falsification --cache
```

### 5.7.2 Results

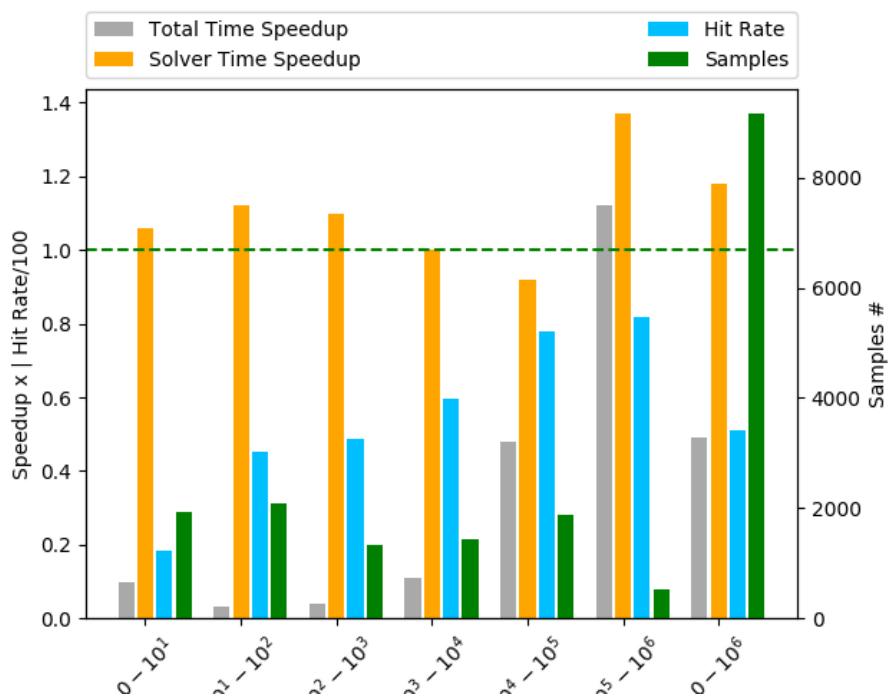
Strat	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
Baseline solver time 0ms to 10ms						
20	203.88	0.10	-0.15	1.06	18.28	1918
22	42.82	0.35	0.16	0.94	9.67	1913
Baseline solver time 11ms to 100ms						
20	6843.60	0.03	-3.84	1.12	45.25	2075
22	862.90	0.24	-0.94	1.03	29.23	2077
Baseline solver time 101ms to 1000ms						
20	36420.09	0.04	-38.27	1.10	48.81	1333
22	8778.42	0.20	-36.98	1.10	28.96	1371
Baseline solver time 1s to 10s						
20	56667.72	0.11	-25.11	1.00	59.81	1432
22	29741.19	0.28	-390.53	1.08	48.75	2123
Baseline solver time 10s to 100s						
20	34894.43	0.48	2629.30	0.92	78.11	1874
22	21812.26	0.61	4611.86	0.87	58.61	2045
Baseline solver time 100s to 1000s						
20	-24593.32	1.12	-60471.45	1.37	81.93	523
22	-17950.08	1.09	-32487.95	1.18	64.74	548
Baseline solver time 0s to 1000s						
20	21498.36	0.49	-2926.76	1.18	51.22	9155
22	11096.49	0.66	-918.29	1.05	37.49	10077

**Table 5.32:** Performance of all strategies over all samples in data set 4

## 5.7. DATA SET 4

101

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	203.88	0.10	-0.15	1.06	18.28	1918
$10^1 - 10^2$	6843.60	0.03	-3.84	1.12	45.25	2075
$10^2 - 10^3$	36420.09	0.04	-38.27	1.10	48.81	1333
$10^3 - 10^4$	56667.72	0.11	-25.11	1.00	59.81	1432
$10^4 - 10^5$	34894.43	0.48	2629.30	0.92	78.11	1874
$10^5 - 10^6$	-24593.32	1.12	-60471.45	1.37	81.93	523
$0 - 10^6$	21498.36	0.49	-2926.76	1.18	51.22	9155

**Table 5.33:** Performance of strategy 20 over all samples in data set 4**Figure 5.27:** Performance of strategy 20 over all samples in data set 4

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	42.82	0.35	0.16	0.94	9.67	1913
$10^1 - 10^2$	862.90	0.24	-0.94	1.03	29.23	2077
$10^2 - 10^3$	8778.42	0.20	-36.98	1.10	28.96	1371
$10^3 - 10^4$	29741.19	0.28	-390.53	1.08	48.75	2123
$10^4 - 10^5$	21812.26	0.61	4611.86	0.87	58.61	2045
$10^5 - 10^6$	-17950.08	1.09	-32487.95	1.18	64.74	548
$0 - 10^6$	11096.49	0.66	-918.29	1.05	37.49	10077

**Table 5.34:** Performance of strategy 22 over all samples in data set 4

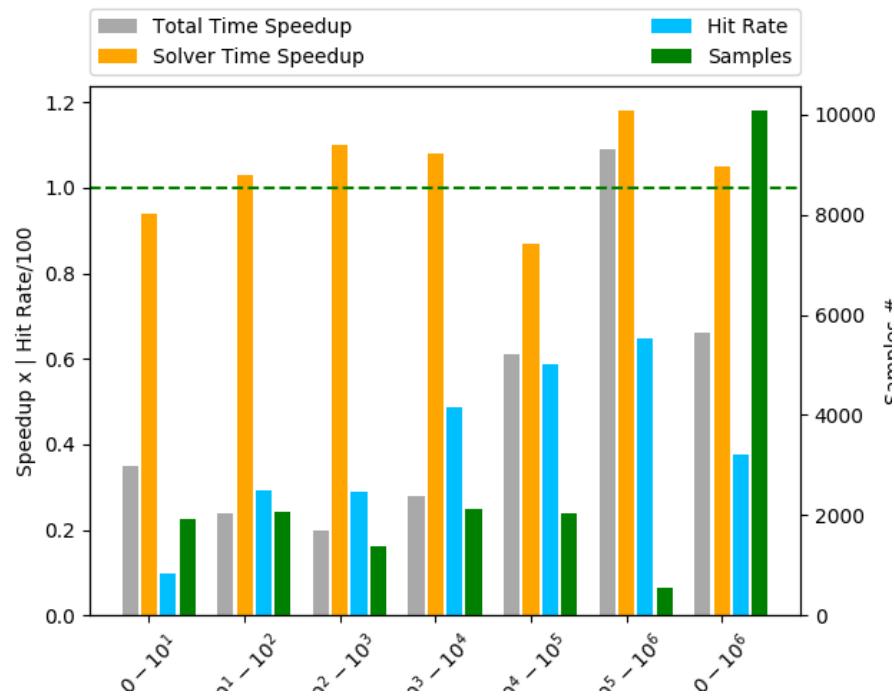


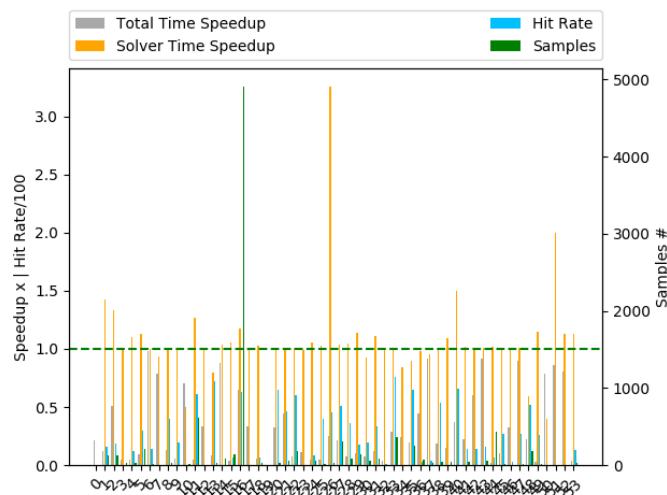
Figure 5.28: Performance of strategy 22 over all samples in data set 4

Label	Folder	TT x	ST x	HR(%)	Samples
0	psyco	0.22	0.00	0.00	1
1	forester-heap	0.12	1.42	15.61	122
2	ldv-linux-3.4-simple	0.51	1.33	18.80	123
3	ssh	0.05	1.01	0.00	28
4	loop-invgen	0.05	1.10	11.76	34
5	ldv-linux-3.14	0.09	1.13	29.93	208
6	reducercommutativity	0.98	1.00	13.69	13
7	ldv-linux-3.7.3	0.79	0.93	0.00	1
8	ldv-regression	0.13	1.00	39.69	35
9	ldv-linux-3.0	0.06	1.01	20.00	1
10	signedintegeroverflow-regression	0.70	0.50	0.00	10
11	seq-mthreaded	0.05	1.27	61.10	620
12	floats-esbmc-regression	0.33	1.00	0.00	2
13	ldv-validator-v0.8	0.09	0.80	71.86	29
14	termination-crafted-lit	0.88	1.03	0.55	89
15	systemc	0.04	1.06	7.08	149
16	eca-rers2012	0.65	1.17	62.78	4905
17	loop-lit	0.33	1.00	0.00	1
18	memsafety-ext	0.06	1.02	7.05	37
19	array-industry-pattern	0.00	0.00	0.00	0
20	ldv-memsafety-bitfields	0.33	0.99	65.19	32
21	ldv-linux-3.12-rc1	0.44	1.00	46.47	64
22	ldv-linux-4.2-rc1	0.07	1.00	59.82	186
23	ntdrivers-simplified	0.11	0.99	0.00	4
24	list-ext-properties	0.05	1.06	8.30	64
25	list-ext2-properties	0.05	1.03	39.70	23
26	seq-pthread	0.25	3.25	45.33	36
27	ldv-linux-3.16-rc1	0.22	1.04	50.71	306
28	termination-crafted	0.08	1.04	35.87	86
29	array-memsafety	0.11	1.14	17.69	147
30	ldv-challenges	0.07	0.92	19.63	60
31	memsafety	0.12	1.11	33.26	84
32	ldv-linux-4.0-rc1-mav	0.03	1.00	1.00	1
33	busybox-1.22.0	0.29	1.00	75.70	360
34	array-examples	0.24	0.84	0.00	2
35	ldv-memsafety	0.20	0.90	64.92	259
36	loops	0.44	0.98	2.68	76
37	recursive	0.91	0.96	3.62	24
38	ldv-validator-v0.6	0.19	1.01	53.34	38
39	ssh-simplified	0.15	1.10	0.00	43
40	ldv-commit-tester	0.37	1.50	66.00	3
41	memsafety-ext2	0.22	1.02	13.78	49
42	pthread	0.61	1.00	14.33	3
43	bitvector	0.91	1.01	15.66	58
44	product-lines	0.02	1.01	6.53	437
45	ldv-consumption	0.11	1.01	26.72	18
46	loop-industry-pattern	0.32	1.01	2.67	3
47	termination-numeric	0.90	1.01	27.18	22
48	recursive-simple	0.22	0.59	51.61	186
49	heap-manipulation	0.03	1.15	25.62	21
50	loop-acceleration	0.79	0.40	0.00	6
51	bitvector-regression	0.86	2.00	0.00	5
52	bitvector-loops	0.80	1.13	0.00	2
53	list-properties	0.04	1.13	12.78	36

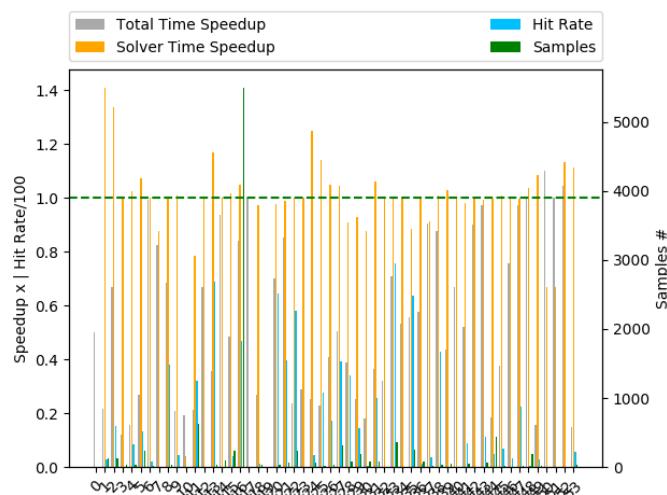
**Table 5.35:** Performance of strategy 20 across all categories for all samples in data set 4

Label	Folder	TT x	ST x	HR(%)	Samples
0	psyco	0.50	0.00	0.00	1
1	forester-heap	0.22	1.41	2.89	132
2	ldv-linux-3.4-simple	0.67	1.33	15.33	132
3	ssh	0.12	1.01	0.00	28
4	loop-invgen	0.16	1.03	8.63	41
5	ldv-linux-3.14	0.27	1.07	13.45	244
6	reducercommutativity	0.99	1.00	2.00	13
7	ldv-linux-3.7.3	0.83	0.88	0.00	1
8	ldv-regression	0.68	1.00	38.03	35
9	ldv-linux-3.0	0.21	1.01	4.50	2
10	signedintegeroverflow-regression	0.19	0.04	0.00	10
11	seq-mthreaded	0.21	0.78	32.14	631
12	floats-esbmc-regression	0.67	1.00	0.00	2
13	ldv-validator-v0.8	0.36	1.17	68.97	30
14	termination-crafted-lit	0.94	1.00	0.00	89
15	systemc	0.48	1.02	4.02	239
16	eca-rers2012	0.84	1.05	46.83	5491
17	loop-lit	1.00	0.00	0.00	1
18	memsafety-ext	0.27	0.97	1.40	42
19	array-industry-pattern	0.00	0.00	0.00	0
20	ldv-memsafety-bitfields	0.70	0.98	64.47	32
21	ldv-linux-3.12-rc1	0.85	0.99	39.53	64
22	ldv-linux-4.2-rc1	0.24	1.00	57.88	237
23	ntdrivers-simplified	0.29	1.00	0.00	4
24	list-ext-properties	0.25	1.25	4.47	70
25	list-ext2-properties	0.23	1.14	27.79	24
26	seq-pthread	0.41	1.05	17.42	38
27	ldv-linux-3.16-rc1	0.50	1.04	39.35	308
28	termination-crafted	0.39	0.91	34.02	86
29	array-memsafety	0.25	0.93	14.43	186
30	ldv-challenges	0.18	0.88	0.51	81
31	memsafety	0.37	1.06	25.57	86
32	ldv-linux-4.0-rc1-mav	0.32	1.00	0.00	2
33	busybox-1.22.0	0.71	1.00	75.70	360
34	array-examples	0.53	1.00	0.00	2
35	ldv-memsafety	0.56	0.89	63.82	259
36	loops	0.58	1.00	1.35	80
37	recursive	0.91	0.91	3.62	24
38	ldv-validator-v0.6	0.88	1.01	42.66	38
39	ssh-simplified	0.44	1.03	0.00	43
40	ldv-commit-tester	0.67	1.00	0.00	3
41	memsafety-ext2	0.52	0.98	8.73	51
42	pthread	0.90	1.00	0.00	3
43	bitvector	0.97	0.99	11.33	58
44	product-lines	0.19	1.00	4.86	447
45	ldv-consumption	0.37	1.01	6.80	25
46	loop-industry-pattern	0.76	1.00	3.20	5
47	termination-numeric	0.97	1.00	22.50	22
48	recursive-simple	1.00	1.04	0.54	186
49	heap-manipulation	0.16	1.08	2.76	25
50	loop-acceleration	1.10	0.67	0.00	6
51	bitvector-regression	1.00	0.67	0.00	5
52	bitvector-loops	1.04	1.13	0.00	2
53	list-properties	0.15	1.11	5.76	38

**Table 5.36:** Performance of strategy 22 across all categories for all samples in data set 4



**Figure 5.29:** Performance of strategy 20 across all categories for all samples in data set 4



**Figure 5.30:** Performance of strategy 22 across all categories for all samples in data set 4



# Chapter 6

## Conclusions

### 6.1 Symbolic Execution vs Bounded Model Checking

During Section 2.4 a list of existing formula caches were mentioned that have shown a performance increase in the field of symbolic execution. The reuse from these caches stretched across runs and showed a clear performance increase for end to end time and always had an increase in performance when it comes to solving time of the SMT solvers.

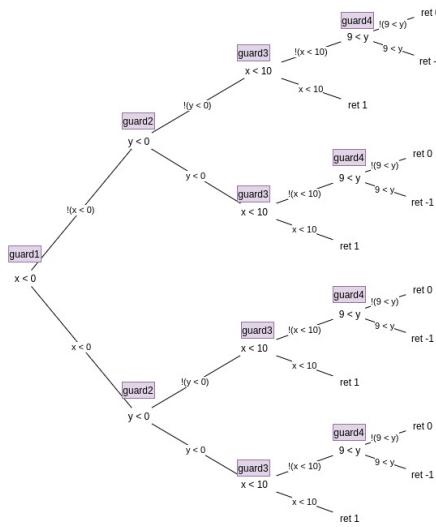
Why do the results for caching in bounded model checking not mirror the results from symbolic execution? To understand the fundamental differences, the following example will be taken from the Green paper [29].

During symbolic execution whenever a branch is found in the program path, both paths of the branch is checked for satisfiability based on the path condition and constraints build up to that point. If either branch is unsatisfiable, that path no longer needs to be explored.

Figure 2.1 shows there are 22 different paths to be explored if symbolic execution is used. The bounded model checking equivalent SMT query would be

```
x = ?
y = ?
x1 = -x
y1 = -y
guard1 = x < 0
x2 = ite guard1 x1 x
guard2 = y < 0
y2 = ite guard2 y1 y
guard3 = x2 < 10
guard4 = 9 < y2
ret1 = 1
ret2 = ite guard3 ret1 -1
ret3 = ite guard4 ret2 0
```

As an example, set the requirement for the verification to be a return result of -1. If 2 solver calls are made at each branch condition by the symbolic execution



**Figure 6.1:** Possible paths to be explored with symbolic execution with guards from bounded model checking displayed

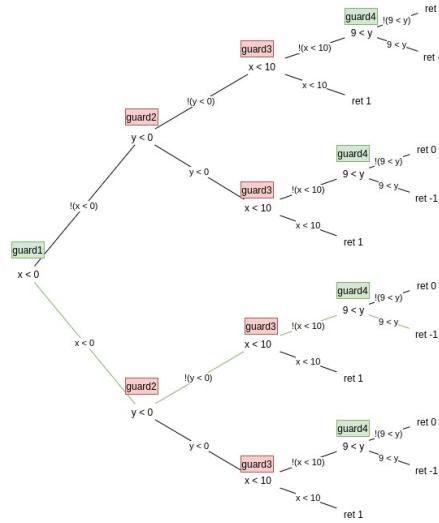
tool then it would take 8 solver calls to reach a return value of  $-1$  along the path  $!(x < 0) \wedge !(y < 0) \wedge !(x < 10) \wedge (9 < y)$ . Figure 6.1 shows all possible paths and the guards that relate to each of these branches from BMC.

To find a path with a return result of  $-1$  from the BMC would constitute a single call where the SMT solver would try to find a path by itself. Figure 6.2 shows guards that are evaluating to True as green and guards evaluating to False as red. The SMT solver must decide what values to assign to each free variable for the path to reach a state that returns  $-1$ .

Here one of the first differences appear between the two approaches. symbolic execution calls the SMT solver multiple times during path exploration, while bounded model checking will only call the SMT solver once, expecting it to figure out what the path is by itself. Fundamentally the question changes from 'is this the path that satisfies the constraints' to 'is there a path that satisfies the constraints'.

When it comes to caching, the approach from symbolic execution affords more opportunities to store results returned by the SMT solvers. In this example the solver could be called a possible 22 times if all branches were explored compared to the single result that would be stored from the bounded model checking approach. This is the second difference in the approaches, as any formula constructed along the path constraints that is found in the cache, will save a solver call, giving an absolute gain for the total amount of solver time at the cost of the lookup. With bounded model checking the best case scenario is removing all assert terms saving a solver call but in most cases the formula is just altered by removing the results of the known assert statement, resulting in an unknown amount of time saved, or in some cases, an increase in time taken as shown in Section 2.5.3.

The third difference is the effectiveness of modifying formulas before they are stored or looked up by the symbolic execution. The process of slicing the formula



**Figure 6.2:** Possible paths to be explored with symbolic execution with guards to force a return value of -1

into smaller formulas based on variables contained in the original formula (variable dependent formulas, Section 4.2) is a lot more effective. Combining slicing with re-ordering of formulas into a canonical form in the Green cache results in just 6 unique formulas compared to the original 22. It is able to do this by modifying formulas like  $x < 0 \wedge x < 10$  into the formula  $x < 0$ . During bounded model checking this simplification cannot be done as it is not clear if the formula  $x < 0 \wedge x < 10$  will evaluate to true during the process to find a satisfiable assignment.

The last difference is the effectiveness to reuse formulas across runs. If the example in listing is slightly modified (mutated) to the following:

```
int m(int x, int y) {
    if (x < 0) x = -x;
    if (y < 0) y = -y;
    if (x < 10) return 1;
    else if (10 < y) return -1;
    else return 0;
}
```

For the case of symbolic execution 8 of the 22 formulas to be evaluated have changed, and of these 8, only 4 are unique that need to be sent to the solver. In the case of bounded model checking, there is only 1 formula in the cache which does not match the current formula, leading to no reuse.

This ability to look at specific execution paths is where most of the gains are made. Fundamentally formulas from bounded model checking require programs to, for a large extent, replicate each others behavior for the given unwind bound. This creates very little reuse possibilities across runs but does allow for some reuse within a single program verification over multiple iterations.

## 6.2 Effectiveness of caching formulas for Bounded Model Checking

Is it effective to cache formulas produced by Bounded Model Checking Tools to be reused again in subsequent iterations? The experimental results analyzed in this document has clearly shown that the same formulas are generated during subsequent iterations of a single verification run. Strategy 20 was shown to be the most effective with regards to hit rate with an average hit rate of 46.51% for data set 1 across 11820 samples, 13.5% for data set 2 across 440 samples, 45.59% for data set 3 across 10494 samples and 51.22% for data set 4 across 9155 samples. The original hypothesis that Bounded Model Checking Tools are constructing formulas to be checked for satisfiability that contain parts previously checked already is indeed shown to be true. Caching of these formulas is thus an effective method of decreasing the size of the formula to be sent to the SMT solver and in some cases avoid calling the SMT solver at all.

## 6.3 Influence of caching formulas produced by BMC on SMT solvers

In Section 2.5.3 it was shown that modifying a formula being sent to an SMT solver can influence the time it takes to determine the satisfiability of the formula both positively and negatively. With two different SMT solvers analyzed in the experiments, did they experience a similar change in effectiveness when handling the smaller formulas? Data set 3 and data set 4 was run with the exact same parameters on the same hardware with the only difference being the SMT solver, data set 3 made use of Z3 and data set 4 of boolector.

Data Set	Strat	HR(%)	ST x	Samples
3	20	45.59	1.22	10494
3	22	30.25	1.01	10697
4	20	51.22	1.18	9155
4	20	37.49	1.05	10077

**Table 6.1:** Performance of strategy 20 and 22 across data set 1 and data set 2

The best indicator of the efficiency of each cache with regards to the smaller formulas is the amount of samples that each strategy completed for their relative data sets. Since the only difference in data sets are the solvers, the exact same results will be found for the effectiveness of the cache. With boolector generally used by ESBMC during competition runs (as it is considered to be more effective) it is perhaps surprising to see that it has less samples to compare. The difference in hit rate can be attributed to the difference in samples and also shows that Z3 is benefiting more from caching, although both SMT solvers are showing net increases in performance with regards to the amount of time it takes to compute the satisfiability of a formula.

## 6.4 Cache influence on ESBMC

Evaluating the different strategies that ESBMC can make use of with regards to caching on formulas within a single verification run did yield promising results. Even though there wasn't always an outright performance increase, there were plenty of encouraging signs that caching can be used effectively. The best strategies were able to decrease the amount of asserts to be evaluated by the SMT solver by nearly 50% with a significant performance hit while other strategies were able to remove closer to 40% of all asserts but at a minor cost in overall performance. The experiments also showed that the cache has room for improvement with regards to the efficiency with which it stores and retrieves the formulas. There were very few occasions where the solver had a measured speedup of more than 1.5 without the entire formula being shown to be unsatisfiable already, in these cases the solver has a speedup of more than 2 with regards to the amount of time it took to compute a satisfiability result (as the solver did not have to be called at all). This is often the case with concurrent formula where the interleavings generated in a single iteration often being the exact same formula, avoiding a solver call. The large amount of interleavings did cause a substantial amount of overhead which made the feasibility of caching these formulas questionable at best.



# Chapter 7

## Future Work

### 7.1 Handling branches individually

The formulas that are generated by ESBMC represent branching conditions with `ite` (if-then-else) terms containing a checking term along with two separate subsequent terms that will be assigned to a given variable depending on the result of the term that is being checked. Two duplicate formulas can be created, one where the branch condition was True and another where it was False. Creating two separate formulas would increase the effectiveness of variable dependent formulas (Section 4.2) as all variables across all branches will no longer be connected due a single bound checks that determined different paths. If both formulas have an unsatisfiable result in the cache then the original formula would collectively still have an unsatisfiable result. If only one is in the cache then other one would still have to be checked. If all branches were split into separate formulas then it would be equivalent to just checking all program paths in the first place, this will lead to the original state space explosion problem but some heuristic could perhaps be identified deciding how many splits should be permitted.

### 7.2 Symbolic Execution and Bounded Model Checking hybrid

Bounded Model Checking attempts to solve the problem of identifying any possible branch that could lead to a property violation. In contrast Symbolic Model Checking explores a single path at a time by calling the solver at every branch it finds. A hybrid strategy could be created where a specific path is still followed with standard Symbolic Model Checking techniques, but once a branch is found all potential paths from there on can be checked for satisfiability by a Bounded Model Checker. This will be useful if no path can be followed further as all paths would be considered during a single SMT solver call. If a single path is satisfiable though for further exploration, it will just return that path and give no further insight into the other paths.

### 7.3 Propagate Assumptions

While producing *SSA Steps*, ESBMC produces assumptions that for the most part indicates that a certain loop reached its upper bound and the variable associated with that upper bound is assumed to have been reached. This assumption is then combined with subsequent assert statements. When the *Split Assumptions from Asserts* (Section 4.8) option is enabled along with *Variable Propagation* (Section 4.8), these assumptions get propagated through to the `ite` (if-then-else) terms. Both branches of the formula are still active though unless *Simplification* (Section 4.7.2) is enabled that could potentially remove the parts of the branch that will not be executed for the specific formula. This leads to smaller formulas that could then lead to more potential matches.

It is often the case however that an assumption is not necessarily just True or False in combination with an assert, but rather that combinations of assumptions all evaluate to True. These combinations can be turned into a disjunct and their values propagated in duplicate formulas. Only the formulas for which there is no match in the cache then needs to be checked for satisfiability.

### 7.4 Increasing cache efficiency

All results obtained for this document were created with a combined key for formula lookup. During these runs no clashes were detected, implying that the `crc` value could have been used by itself as the key for an increase in performance, further evaluations should be done to see the impact of changing the key. Although there were no clashes detected for the keys in these experiments while using the cache for single verification runs, it is possible that clashes will start appearing once the cache is persistent across multiple verification runs, an analysis would be needed to determine if clashes become more likely at that point and how often it happens. It was also clearly shown that the caching process for strategy 20 was the dominating factor of the overhead, this area should be explored first to ensure the most efficient data structures are used.

### 7.5 Solving modified formulas

The cache already supplies an option to the user by which the cache will then send the modified formulas to the SMT solver. Enabling this option still needs to be evaluated as some of the work being done during the formula modifications also get done by the SMT solver like variable propagation. Avoiding the duplicate work might help speed up the SMT solver and further evaluation needs to be done on this area.

### 7.6 Persistent Formula Cache and Canonical Form

Small evaluations of a persistent cache were run for the current implementation of the cache without the results being added. The implementation relied on a bound that must be reachable by all files to be evaluated and they were then checked for satisfiability in sequence without killing the ESBMC process as soon as a verification run was completed. Initial findings showed very little reuse but a full evaluation should be done with a cache that does not form part of ESBMC but rather keeps the results persistent through some other means (like redis).

The current implementation does not fully implement a canonical form but rather just small modifications. This allowed for some of the assumptions needed by the filters with regards to the order in which terms are expected to appear. Further investigating is needed to determine if there are greater reuse opportunities to be gained from a more standard canonical form and how the filters would have to be modified to accommodate these changes.

## 7.7 Heuristically matching solution spaces

Utopia [2], another formula cache that has been used in conjunction with Symbolic Model Checking describes a technique of finding 10 formulas, 5 from a cache with only satisfiable formulas and 5 more from a cache with only unsatisfiable formulas. These are the only formulas that are then compared to the original formula to determine if there is already a known satisfiability result for the formula. This cache implements the idea of subset-matching but compares the formula to a potentially large set of possible matches. A similar heuristic could perhaps be implemented to narrow down the possible matches even further.



# Bibliography

- [1] Andrea Aquino, Francesco A. Bianchi, Meixian Chen, Giovanni Denaro, and Mauro Pezzè. Reusing constraint proofs in program analysis. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 305–315, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771802. URL <http://doi.acm.org/10.1145/2771783.2771802>.
- [2] Andrea Aquino, Giovanni Denaro, and Mauro Pezzè. Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE ’17, pages 427–437, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.46. URL <https://doi.org/10.1109/ICSE.2017.46>.
- [3] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.
- [4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). [www.SMT-LIB.org](http://www.SMT-LIB.org), 2016.
- [5] Shoham Ben-David, Cindy Eisner, Daniel Geist, and Yaron Wolfsthal. Model checking at ibm. *Formal Methods in System Design*, 22(2):101–108, 2003.
- [6] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009. ISBN 1586039296, 9781586039295.
- [7] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic Model Checking without BDDs*, pages 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-49059-3. doi: 10.1007/3-540-49059-0\_14. URL [https://doi.org/10.1007/3-540-49059-0\\_14](https://doi.org/10.1007/3-540-49059-0_14).
- [8] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. volume 58 of *Advances in Computers*, pages 117 – 148. Elsevier, 2003. doi: [https://doi.org/10.1016/S0065-2458\(03\)58003-2](https://doi.org/10.1016/S0065-2458(03)58003-2). URL <http://www.sciencedirect.com/science/article/pii/S0065245803580032>.
- [9] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays, 03 2009.

- [10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142 – 170, 1992. ISSN 0890-5401. doi: [https://doi.org/10.1016/0890-5401\(92\)90017-A](https://doi.org/10.1016/0890-5401(92)90017-A). URL <http://www.sciencedirect.com/science/article/pii/089054019290017A>.
- [11] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003. ISSN 0004-5411. doi: 10.1145/876638.876643. URL <http://doi.acm.org/10.1145/876638.876643>.
- [12] Edmund Clarke, Daniel Kroening, and Flavio Lerda. *A Tool for Checking ANSI-C Programs*, pages 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24730-2. doi: 10.1007/978-3-540-24730-2\_15. URL [https://doi.org/10.1007/978-3-540-24730-2\\_15](https://doi.org/10.1007/978-3-540-24730-2_15).
- [13] Edmund M. Clarke. *The Birth of Model Checking*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-69850-0. doi: 10.1007/978-3-540-69850-0\_1. URL [https://doi.org/10.1007/978-3-540-69850-0\\_1](https://doi.org/10.1007/978-3-540-69850-0_1).
- [14] Edmund M. Clarke and E. Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*, pages 52–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982. ISBN 978-3-540-39047-3. doi: 10.1007/BFb0025774. URL <https://doi.org/10.1007/BFb0025774>.
- [15] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, September 1994. ISSN 0164-0925. doi: 10.1145/186025.186051. URL <http://doi.acm.org/10.1145/186025.186051>.
- [16] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35746-6. doi: 10.1007/978-3-642-35746-6\_1. URL [https://doi.org/10.1007/978-3-642-35746-6\\_1](https://doi.org/10.1007/978-3-642-35746-6_1).
- [17] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999. ISBN 9780262032704. URL <https://books.google.co.za/books?id=Nmc4wEaLXFEC>.
- [18] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE ’09, pages 137–148, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. doi: 10.1109/ASE.2009.63. URL <http://dx.doi.org/10.1109/ASE.2009.63>.
- [19] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. *Verifying temporal properties of sequential machines without building their state diagrams*, pages 23–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. ISBN 978-3-540-38394-9. doi: 10.1007/BFb0023716. URL <https://doi.org/10.1007/BFb0023716>.

- [20] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL <http://doi.acm.org/10.1145/115372.115320>.
- [21] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.
- [22] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 177–187, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771806. URL <http://doi.acm.org/10.1145/2771783.2771806>.
- [23] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [24] Aina Niemetz, Mathias Preiner, and Armin Biere. Turbo-charging lemmas on demand with don’t care reasoning. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD ’14, pages 29:179–29:186, Austin, TX, 2014. FMCAD Inc. ISBN 978-0-9835678-4-4. URL <http://dl.acm.org/citation.cfm?id=2682923.2682955>.
- [25] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. 2015.
- [26] Susan Owicky and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, May 1976. ISSN 0001-0782. doi: 10.1145/360051.360224. URL <http://doi.acm.org/10.1145/360051.360224>.
- [27] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.
- [28] M Preiner, A Niemetz, and Armin Biere. Lemmas on demand for lambdas. 1130: 28–37, 01 2014.
- [29] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE ’12, pages 58:1–58:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393665. URL <http://doi.acm.org/10.1145/2393596.2393665>.

# Appendices

# **Appendix A**

# **Experiment Results**

## A.1 Experiment 1 Results

### Setup

The following commands are used for this experiment:

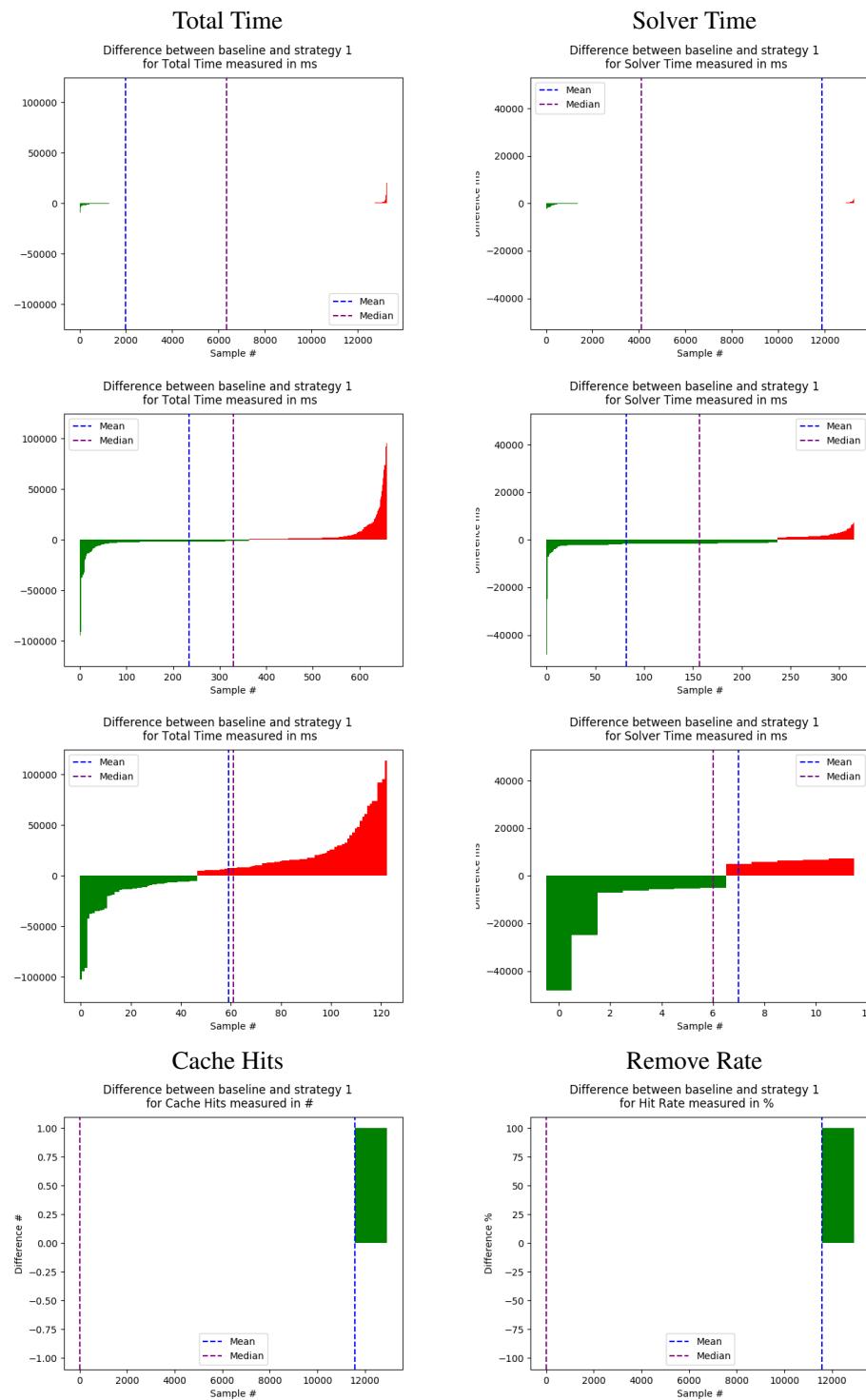
```
--cache --strat1
or
--cache --combine-asserts --lookup-crc-full
```

### Data Set 1 Results

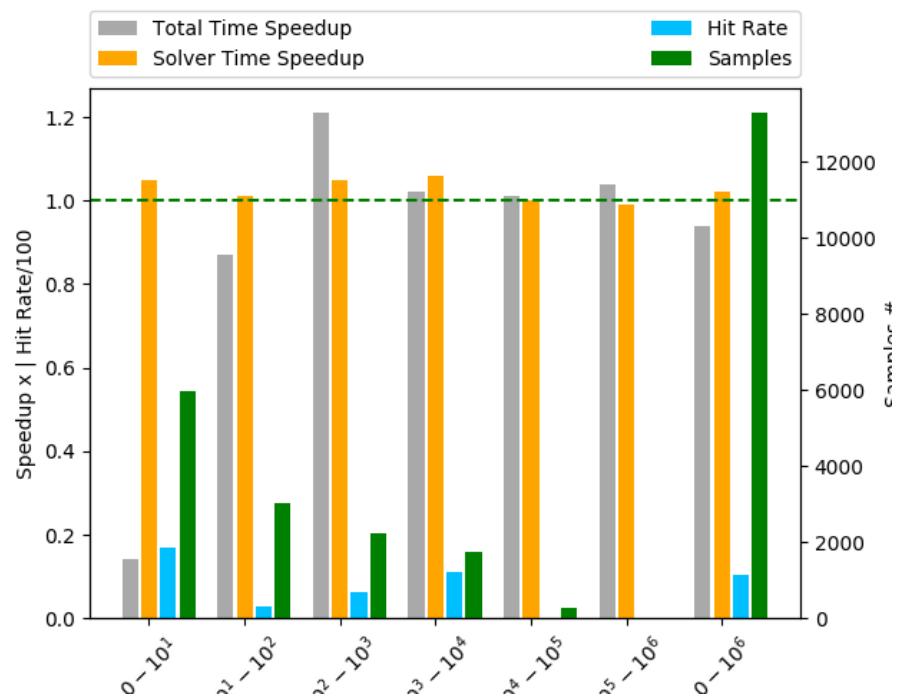
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	1106500	-102423	113512	83.32	1.0	13280	7455	4181
1000	785192	-102423	113512	1189.68	-1218.5	660	297	363
5000	978805	-102423	113512	7957.76	7593.0	123	76	47
Solver Time (ms)								
0	-349055	-48305	7177	-26.28	0.0	13280	4119	5183
1000	-328495	-48305	7177	-1039.54	-1490.0	316	79	237
5000	-71193	-48305	7177	-5932.75	-5263.0	12	5	7
Hit Rate (%)								
0	136300	0	100	10.53	0.0	13280	1363	0
Cache Hits (#)								
0	1363	0	1	0.11	0.0	13280	1363	0

## A.1. EXPERIMENT 1 RESULTS

123



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	267.25	0.14	-0.13	1.05	16.89	5963
$10^1 - 10^2$	17.03	0.87	-0.34	1.01	2.74	3032
$10^2 - 10^3$	-116.72	1.21	-17.21	1.05	6.18	2248
$10^3 - 10^4$	-81.72	1.02	-164.12	1.06	10.91	1751
$10^4 - 10^5$	-346.15	1.01	-120.78	1.00	0.00	279
$10^5 - 10^6$	-5239.14	1.04	1780.43	0.99	0.00	7
$0 - 10^6$	83.32	0.94	-26.28	1.02	10.53	13280



## A.2 Experiment 2 Results

### Setup

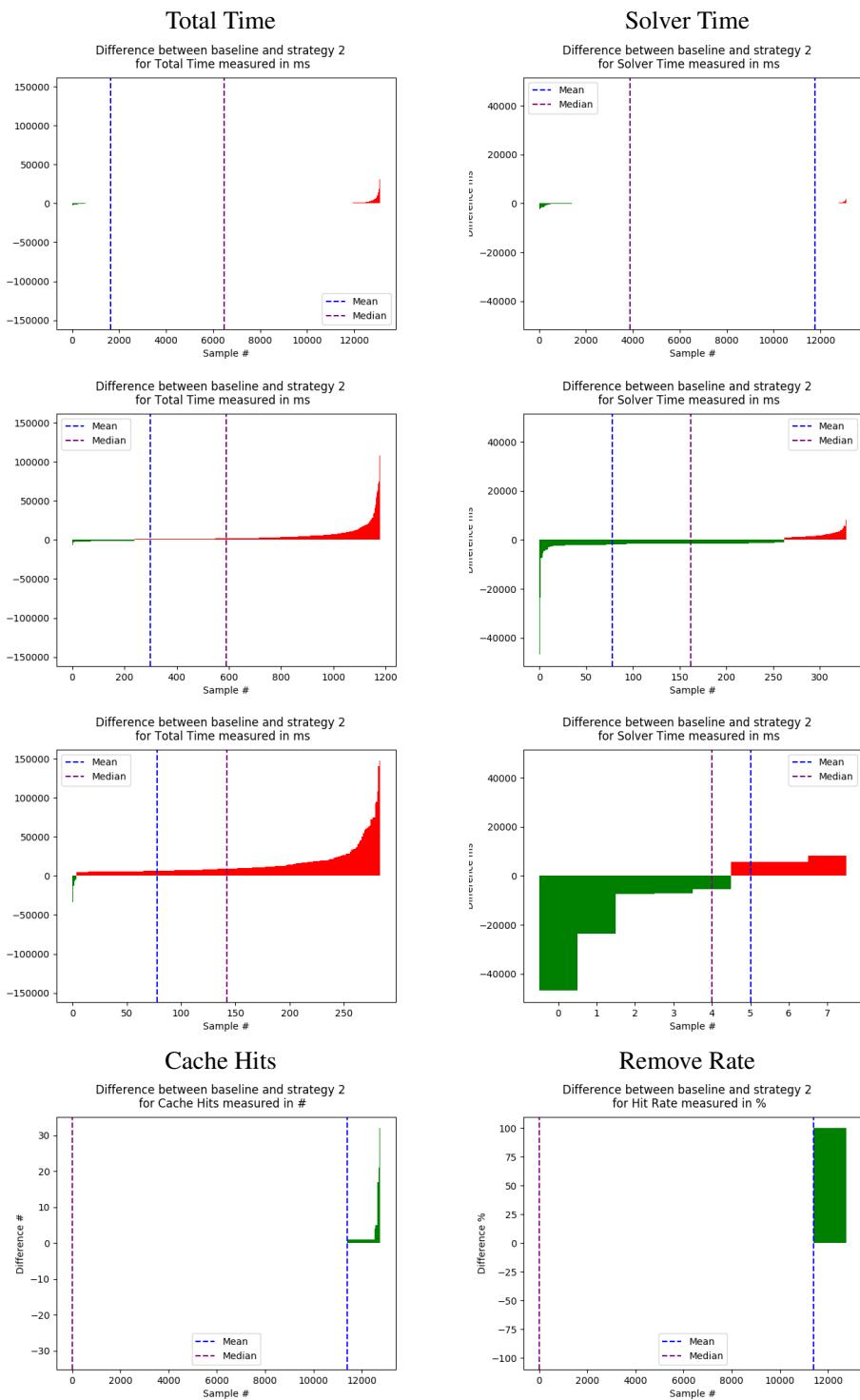
For this experiment, the asserts will be processed separately when looked up and stored in the cache.

The following commands are used for this experiment:

```
--cache --strat2
or
--cache --lookup --lookup-crc-full
```

### Data Set 1 Results

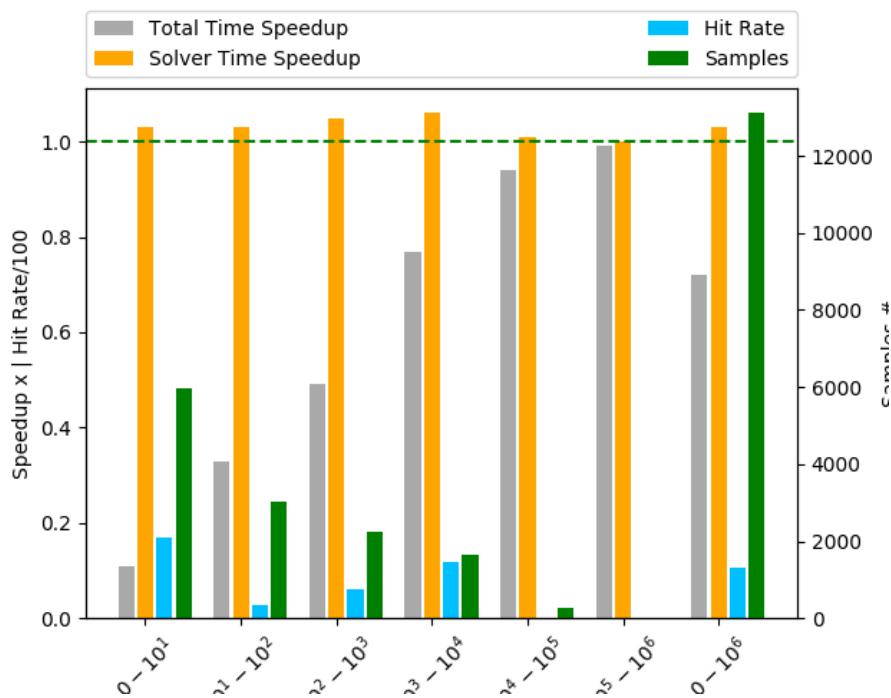
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	6238293	-33638	147222	475.81	6.0	13111	9035	2432
1000	5359105	-33638	147222	4541.61	1787.5	1180	940	240
5000	4371025	-33638	147222	15390.93	8905.5	284	280	4
Solver Time (ms)								
0	-420631	-46871	8164	-32.08	0.0	13111	3886	5239
1000	-391012	-46871	8164	-1188.49	-1495.0	329	67	262
5000	-70869	-46871	8164	-8858.63	-6250.5	8	3	5
Hit Rate (%)								
0	136300	0	100	10.67	0.0	13111	1363	0
Cache Hits (#)								
0	4531	0	32	0.35	0.0	13111	1363	0



## A.2. EXPERIMENT 2 RESULTS

127

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	335.33	0.11	-0.08	1.03	16.91	5956
$10^1 - 10^2$	182.74	0.33	-0.99	1.03	2.75	3018
$10^2 - 10^3$	651.81	0.49	-16.46	1.05	6.21	2239
$10^3 - 10^4$	1088.28	0.77	-180.68	1.06	11.70	1633
$10^4 - 10^5$	1731.40	0.94	-316.22	1.01	0.00	259
$10^5 - 10^6$	763.17	0.99	-555.83	1.00	0.00	6
$0 - 10^6$	475.81	0.72	-32.08	1.03	10.67	13111



### A.3 Experiment 3 Results

#### Setup

The following commands are used for this experiment:

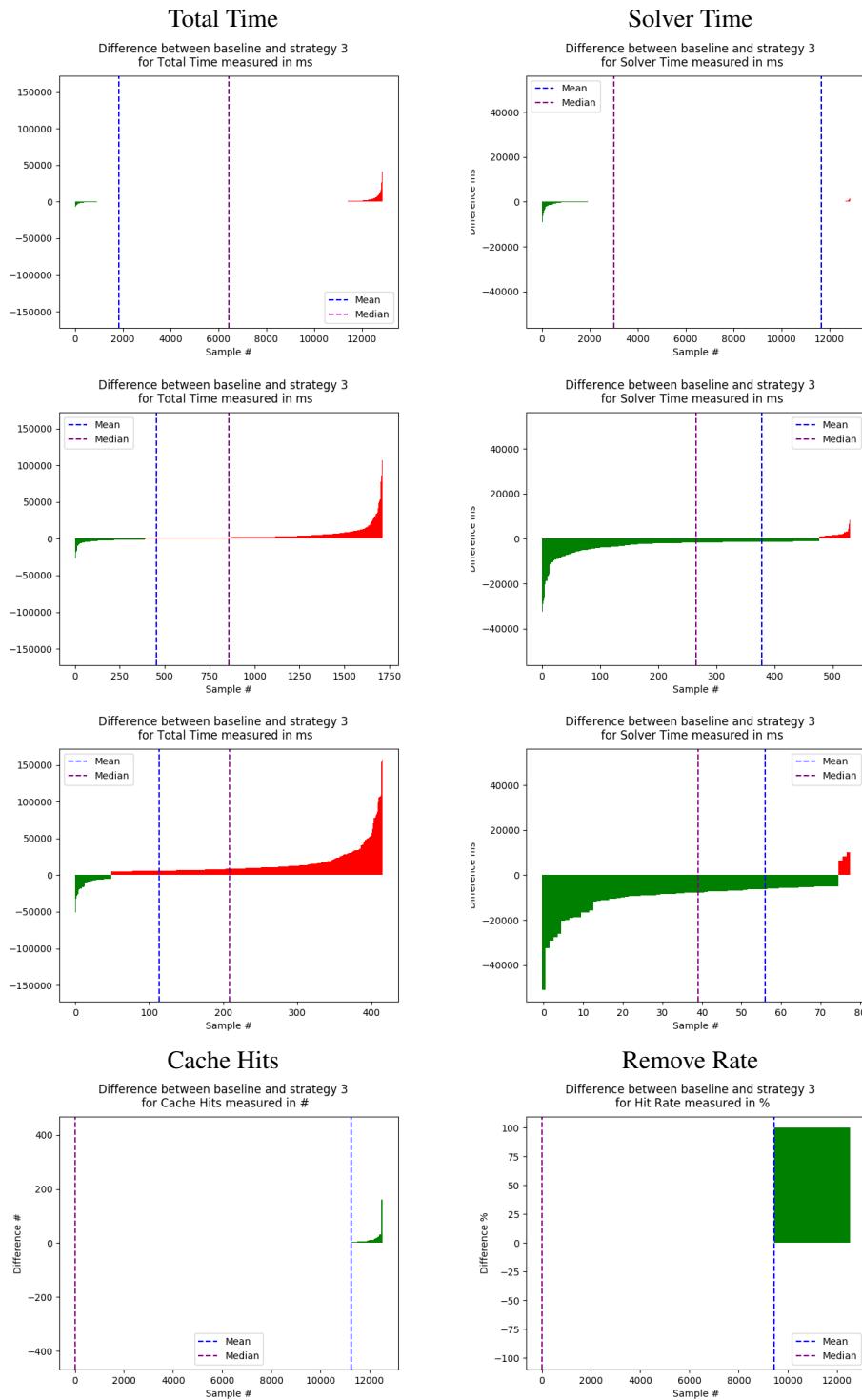
```
--cache --strat3
or
--cache --lookup --lookup-crc-full --variable-renaming
```

#### Data Set 1 Results

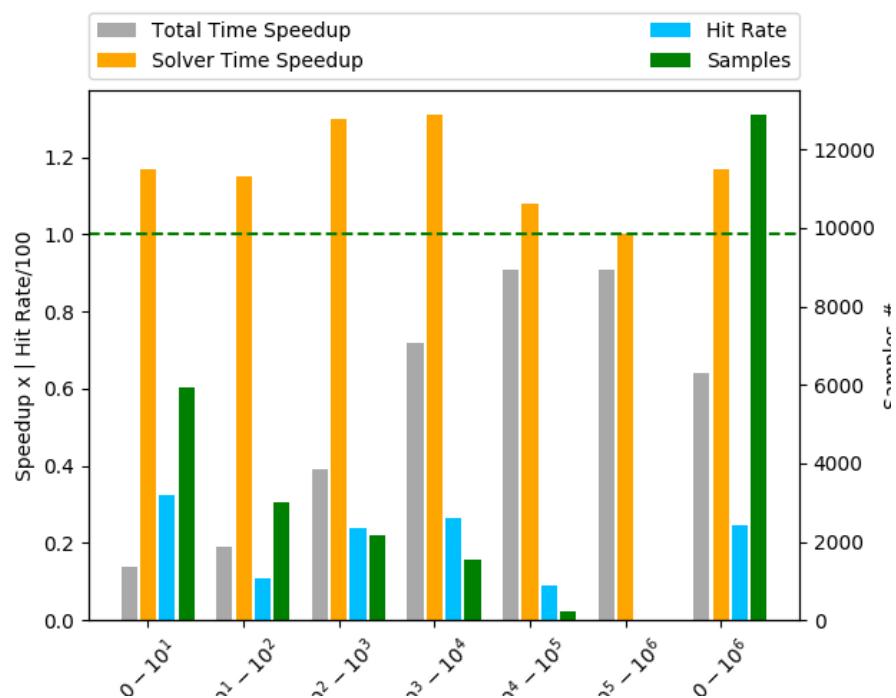
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	7467042	-50840	156742	579.6	8.0	12883	8848	2685
1000	6661688	-50840	156742	3884.37	1752.0	1715	1323	392
5000	5420516	-50840	156742	13030.09	8523.5	416	367	49
Solver Time (ms)								
0	-1712211	-51198	10260	-132.9	0.0	12883	3018	6174
1000	-1496859	-51198	10260	-2818.94	-1641.0	531	53	478
5000	-754215	-51198	10260	-9669.42	-7758.5	78	3	75
Hit Rate (%)								
0	309500	0	100	24.67	0.0	12883	3095	0
Cache Hits (#)								
0	27528	0	426	2.19	0.0	12883	3095	0

## A.3. EXPERIMENT 3 RESULTS

129



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	411.59	0.14	-0.34	1.17	32.44	5950
$10^1 - 10^2$	330.47	0.19	-5.11	1.15	10.92	2994
$10^2 - 10^3$	692.37	0.39	-83.03	1.30	23.93	2173
$10^3 - 10^4$	1238.32	0.72	-718.48	1.31	26.45	1539
$10^4 - 10^5$	2499.88	0.91	-1854.77	1.08	9.01	222
$10^5 - 10^6$	12680.60	0.91	613.40	1.00	0.00	5
$0 - 10^6$	579.60	0.64	-132.90	1.17	24.67	12883



## A.4 Experiment 4 Results

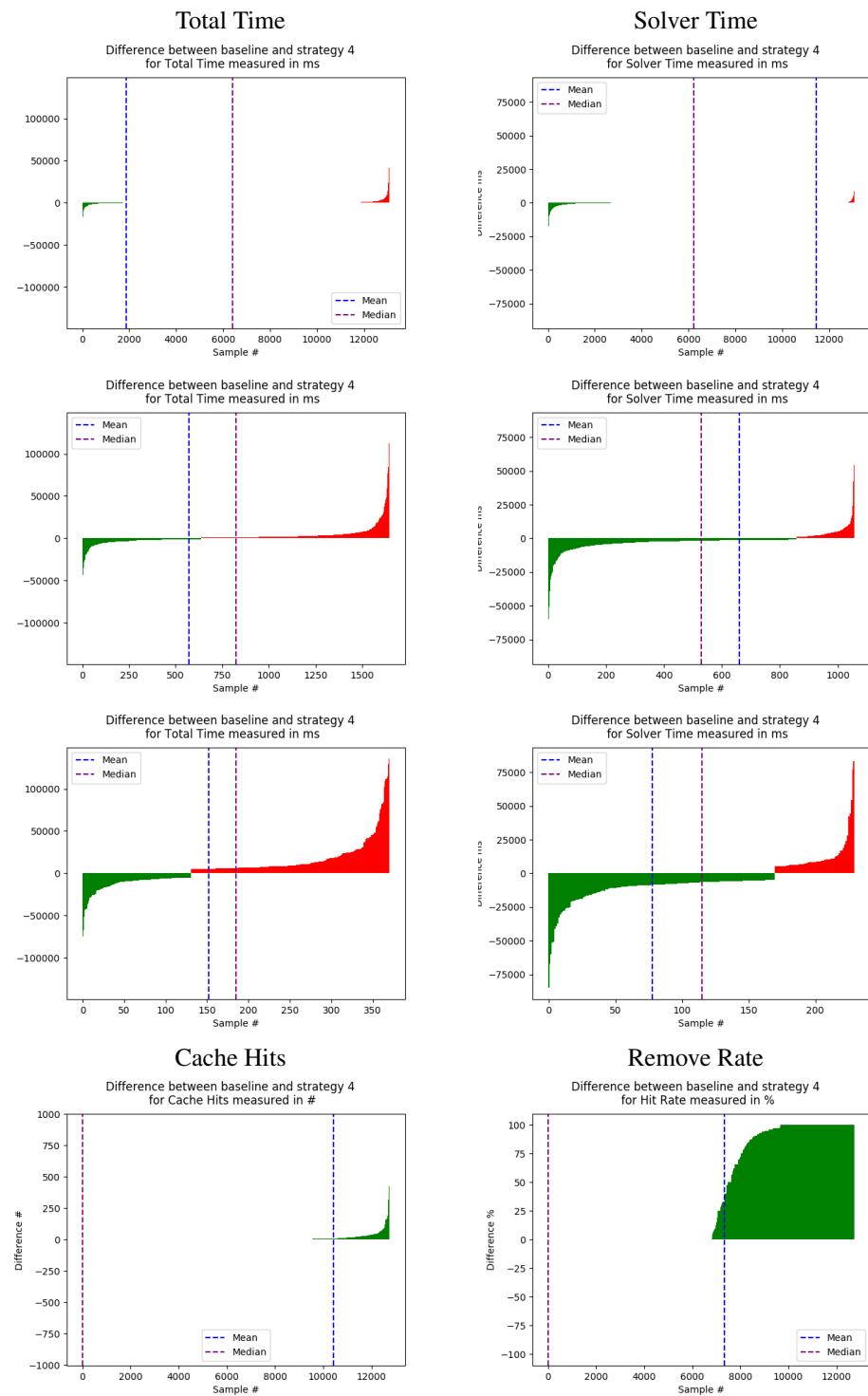
### Setup

The following commands are used for this experiment:

```
--cache --strat4
or
--cache --lookup --lookup-crc-full --variable-renaming --lookup-when-assert
```

### Data Set 1 Results

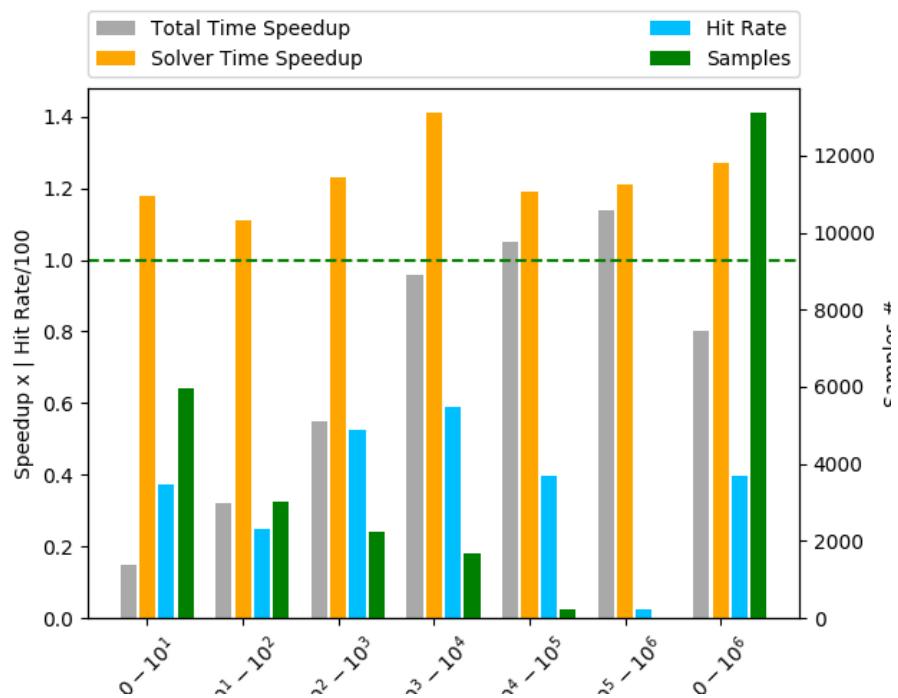
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	3614196	-74791	135629	275.87	3.0	13101	7914	3714
1000	3264621	-74791	135629	1980.96	1303.5	1648	1012	636
5000	2806285	-74791	135629	7564.11	6318.0	371	240	131
Solver Time (ms)								
0	-2789343	-85056	83400	-212.91	-1.0	13101	2724	6872
1000	-2378499	-85056	83400	-2245.99	-1657.0	1059	201	858
5000	-1239662	-85056	83400	-5389.83	-6588.5	230	60	170
Hit Rate (%)								
0	508654	0	100	39.85	0.0	13101	5948	0
Cache Hits (#)								
0	127148	0	913	9.96	0.0	13101	5950	0



## A.4. EXPERIMENT 4 RESULTS

133

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	396.98	0.15	-0.37	1.18	37.15	5955
$10^1 - 10^2$	183.28	0.32	-4.04	1.11	25.11	3011
$10^2 - 10^3$	373.26	0.55	-67.38	1.23	52.57	2224
$10^3 - 10^4$	154.80	0.96	-904.24	1.41	58.78	1664
$10^4 - 10^5$	-1237.95	1.05	-4119.22	1.19	39.82	241
$10^5 - 10^6$	-15177.83	1.14	-21291.33	1.21	2.50	6
$0 - 10^6$	275.87	0.80	-212.91	1.27	39.85	13101



## A.5 Experiment 5 Results

### Setup

The following commands are used for this experiment:

```
--cache --strat5
```

or

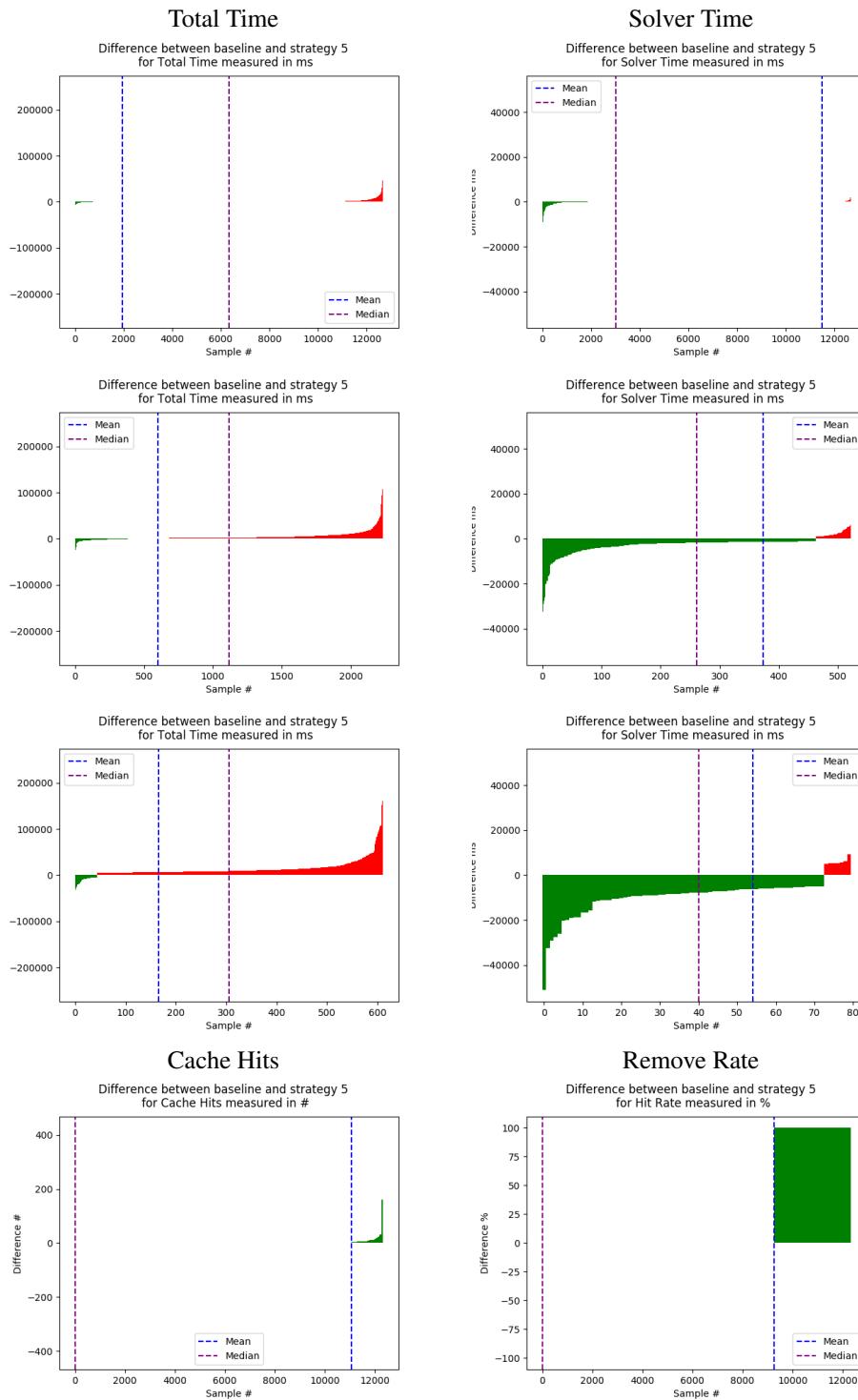
```
--cache --lookup --lookup-crc-full --variable-renaming --variable-dependent
```

### Data Set 1 Results

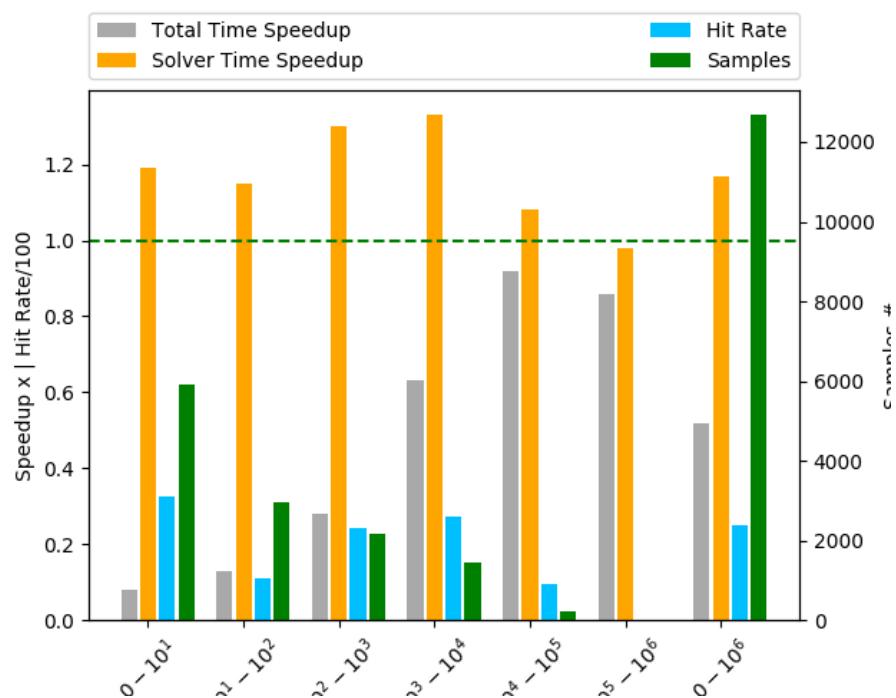
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	11205633	-50669	249567	884.0	13.0	12676	9366	2142
1000	10348474	-50669	249567	4632.26	2292.0	2234	1851	383
5000	8192990	-50669	249567	13387.24	8701.5	612	568	44
Solver Time (ms)								
0	-1642735	-51198	9137	-129.59	0.0	12676	3008	5971
1000	-1443386	-51198	9137	-2759.82	-1697.0	523	60	463
5000	-731531	-51198	9137	-9144.14	-7848.5	80	7	73
Hit Rate (%)								
0	308500	0	100	25.0	0.0	12676	3085	0
Cache Hits (#)								
0	25802	0	426	2.09	0.0	12676	3085	0

## A.5. EXPERIMENT 5 RESULTS

135



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	699.09	0.08	-0.39	1.19	32.73	5899
$10^1 - 10^2$	510.37	0.13	-5.29	1.15	11.09	2948
$10^2 - 10^3$	1086.14	0.28	-82.45	1.30	24.14	2154
$10^3 - 10^4$	1841.24	0.63	-736.77	1.33	27.17	1465
$10^4 - 10^5$	2216.00	0.92	-1827.12	1.08	9.71	206
$10^5 - 10^6$	20926.25	0.86	2124.50	0.98	0.00	4
$0 - 10^6$	884.00	0.52	-129.59	1.17	25.00	12676



## A.6 Experiment 6 Results

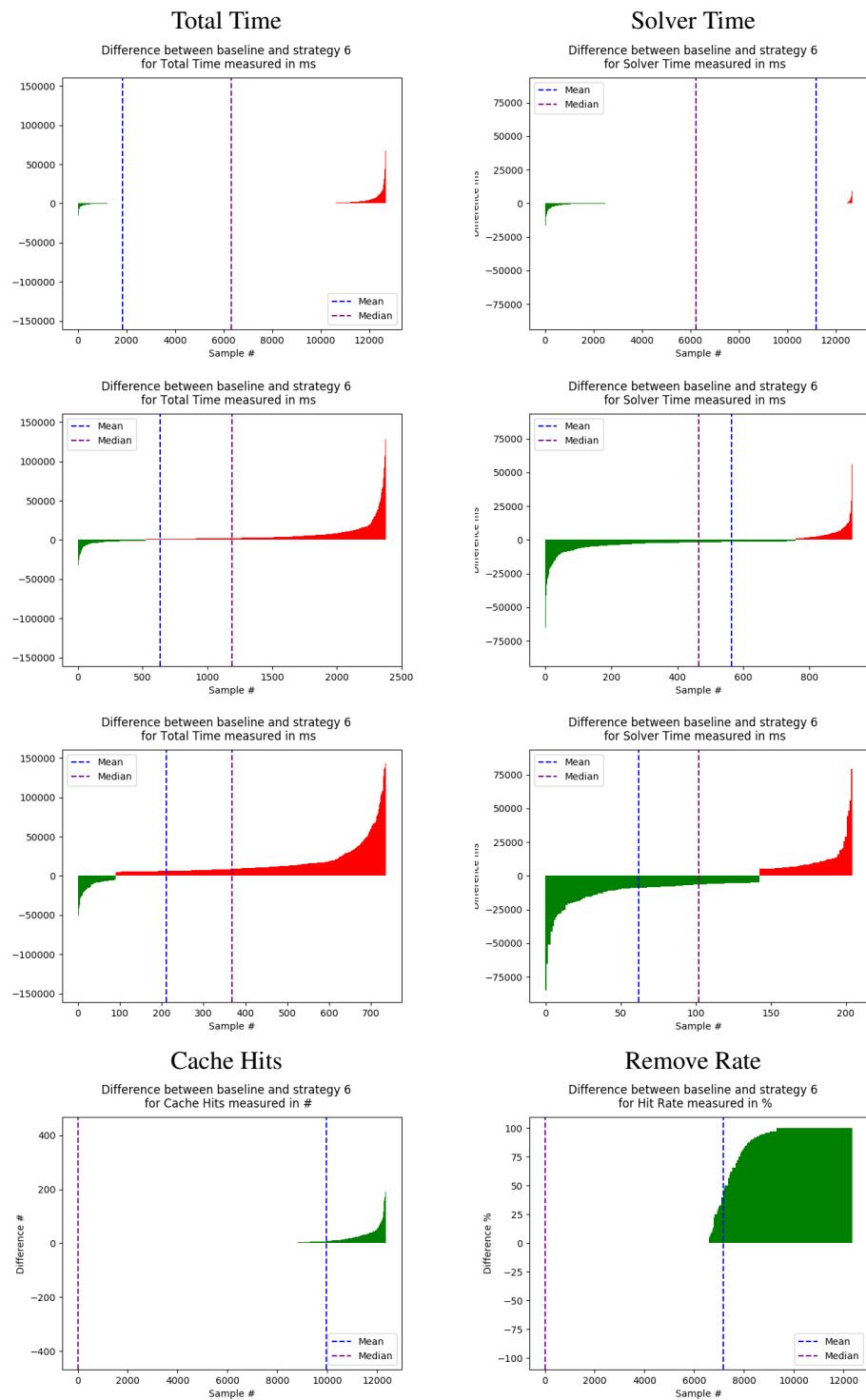
### Setup

The following commands are used for this experiment:

```
--cache --strat6
or
--cache --lookup --lookup-crc-full --variable-renaming
--lookup-when-assert --variable-dependent
```

### Data Set 1 Results

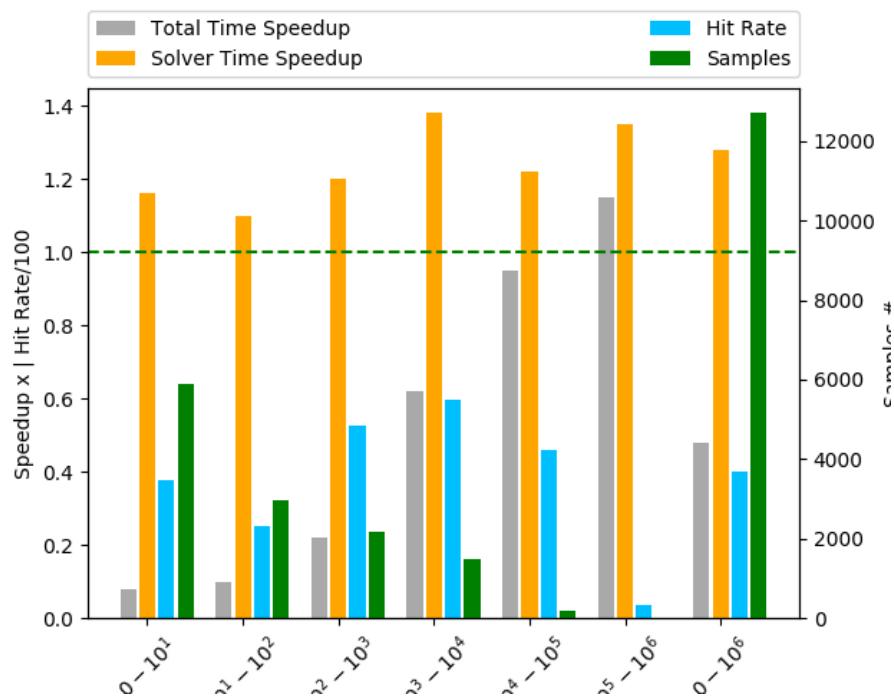
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	12743404	-65440	146372	1003.18	11.0	12703	8993	2511
1000	12117907	-65440	146372	5087.28	2289.0	2382	1859	523
5000	10228110	-65440	146372	13878.03	8666.0	737	646	91
Solver Time (ms)								
0	-2425122	-85308	79213	-190.91	-1.0	12703	2679	6476
1000	-2047294	-85308	79213	-2199.03	-1746.0	931	172	759
5000	-969857	-85308	79213	-4731.01	-6318.0	205	62	143
Hit Rate (%)								
0	494752	0	100	40.01	0.0	12703	5780	0
Cache Hits (#)								
0	93464	0	426	7.56	0.0	12703	5783	0



## A.6. EXPERIMENT 6 RESULTS

139

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	730.91	0.08	-0.33	1.16	37.52	5898
$10^1 - 10^2$	662.84	0.10	-3.62	1.10	25.35	2952
$10^2 - 10^3$	1533.37	0.22	-60.51	1.20	52.56	2164
$10^3 - 10^4$	1981.79	0.62	-840.92	1.38	59.51	1485
$10^4 - 10^5$	1414.51	0.95	-4495.44	1.22	45.93	200
$10^5 - 10^6$	-17070.25	1.15	-33432.00	1.35	3.75	4
$0 - 10^6$	1003.18	0.48	-190.91	1.28	40.01	12703



## A.7 Experiment 7 Results

### Setup

The following commands are used for this experiment:

```
--cache --strat7
```

or

```
--cache --lookup --lookup-crc-full --variable-renaming --lookup-when-assert
```

```
--canonize --variable-simplification --variable-propagation
```

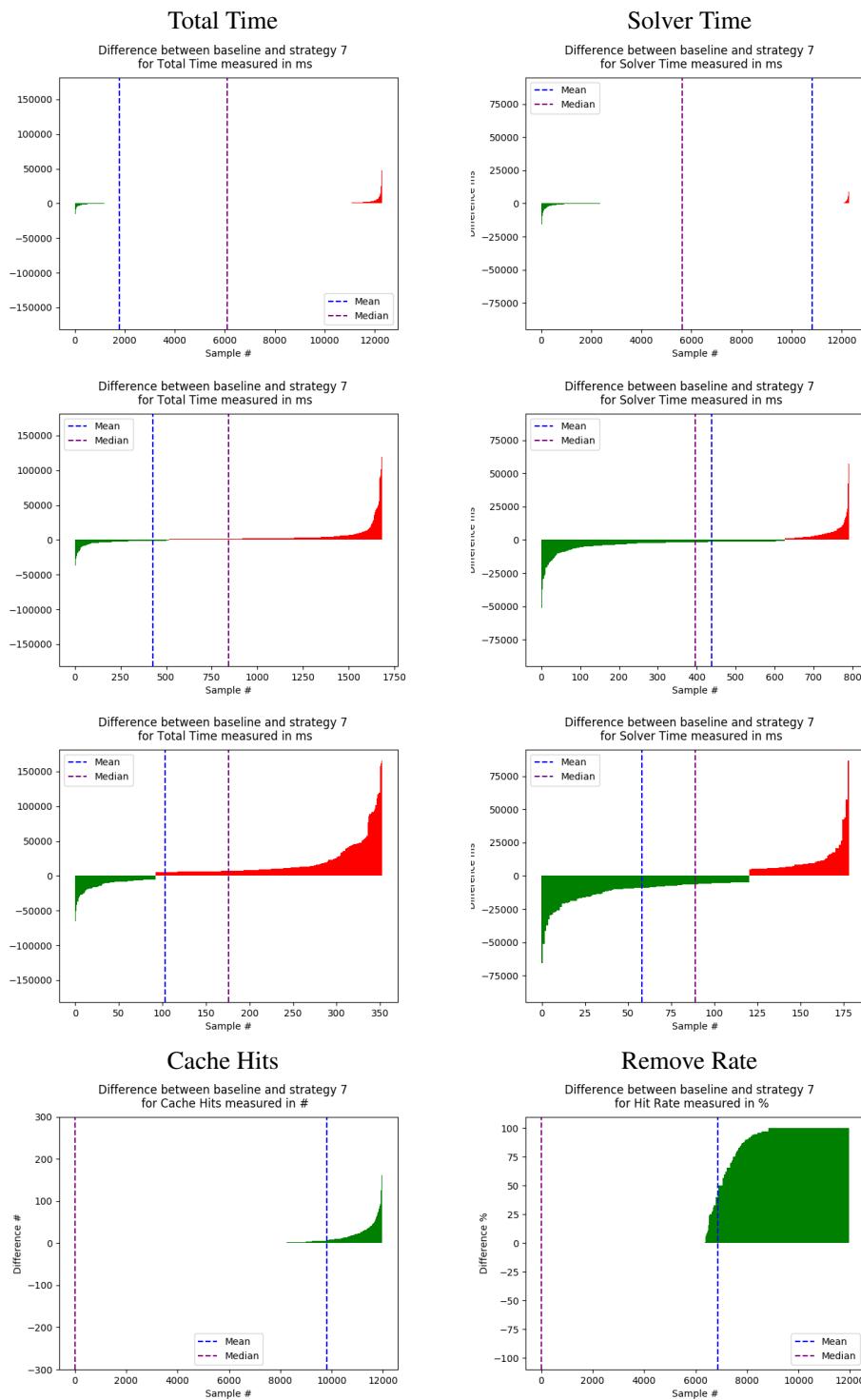
```
--propagation-lookup-crc-full --simplify-replaced
```

### Data Set 1 Result

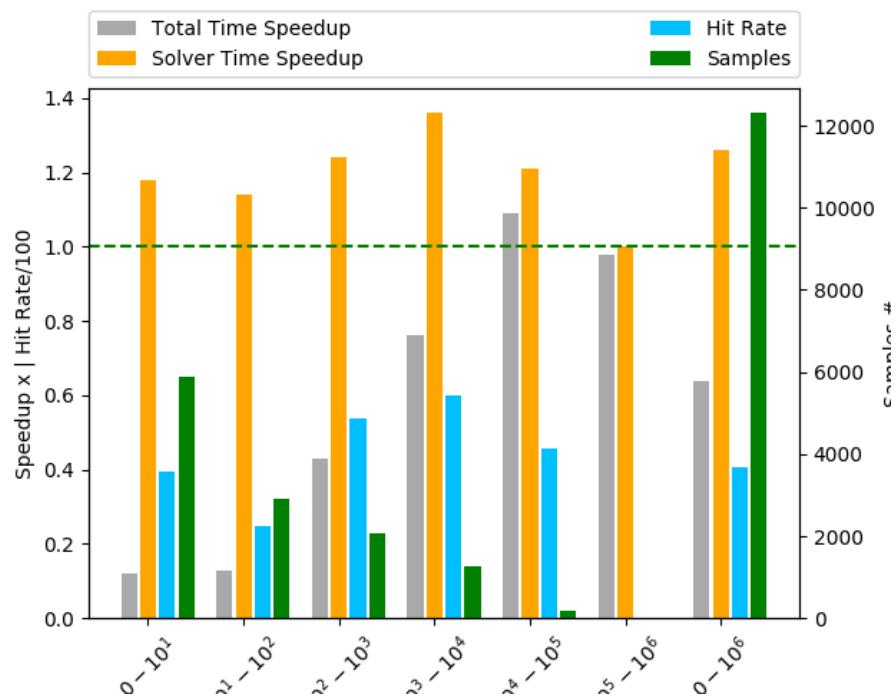
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	5841689	-65722	165219	474.47	10.0	12312	8693	2502
1000	5186936	-65722	165219	3080.13	1634.0	1684	1167	517
5000	4155147	-65722	165219	11770.95	7089.0	353	260	93
Solver Time (ms)								
0	-1958040	-65850	86398	-159.04	-1.0	12312	2137	6683
1000	-1558281	-65850	86398	-1965.05	-1645.0	793	166	627
5000	-710726	-65850	86398	-3970.54	-6241.0	179	58	121
Hit Rate (%)								
0	486223	0	100	40.61	0.0	12312	5606	0
Cache Hits (#)								
0	73199	0	273	6.11	0.0	12312	5606	0

## A.7. EXPERIMENT 7 RESULTS

141



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	450.59	0.12	-0.37	1.18	39.46	5887
$10^1 - 10^2$	446.85	0.13	-4.67	1.14	24.91	2922
$10^2 - 10^3$	530.85	0.43	-68.16	1.24	53.81	2065
$10^3 - 10^4$	941.81	0.76	-778.11	1.36	59.85	1254
$10^4 - 10^5$	-2204.90	1.09	-4535.45	1.21	45.54	182
$10^5 - 10^6$	3703.50	0.98	-148.00	1.00	0.00	2
$0 - 10^6$	474.47	0.64	-159.04	1.26	40.61	12312



## A.8 Experiment 8 Results

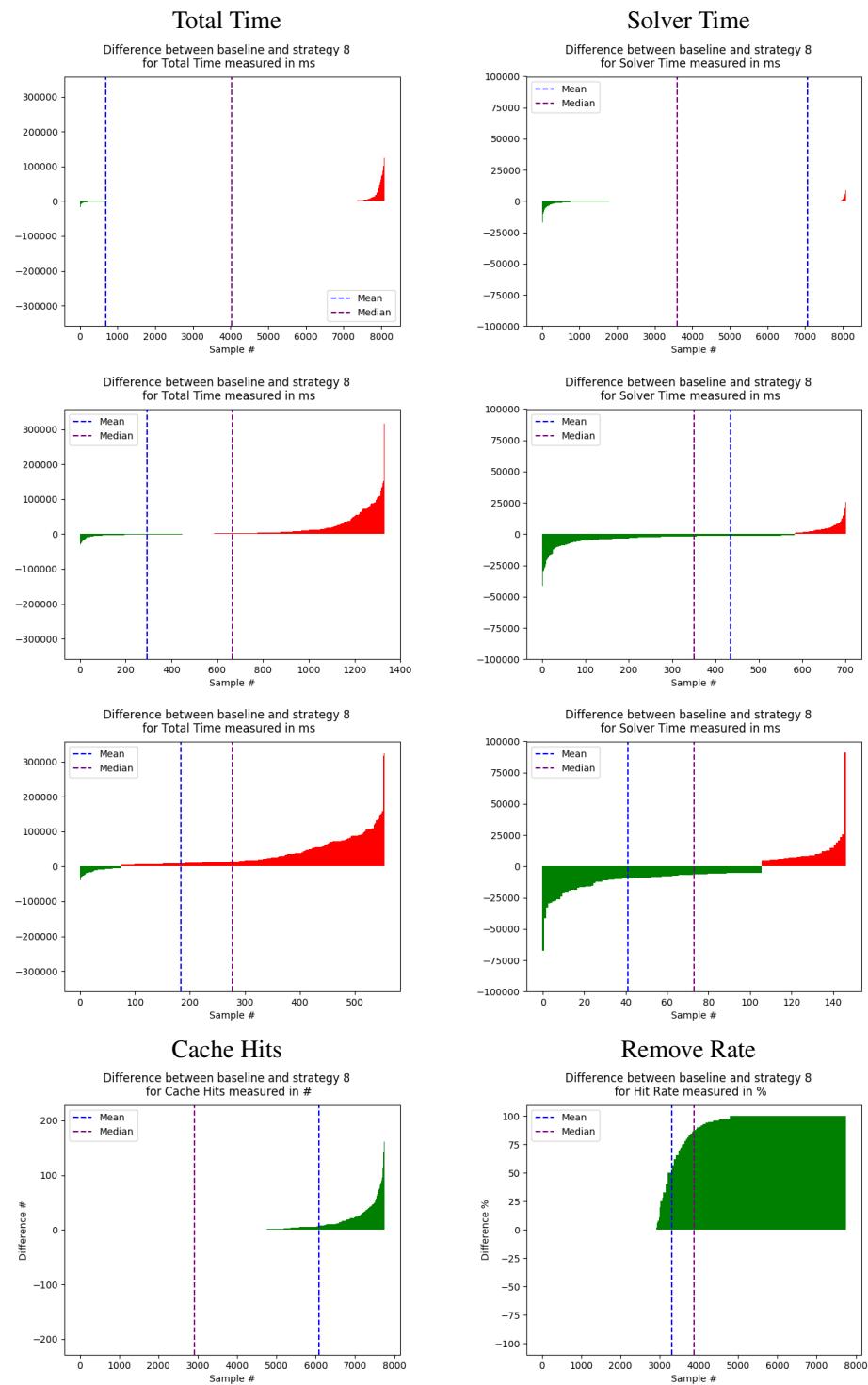
### Setup

The following commands are used for this experiment:

```
--cache --strat8
or
--cache --lookup --lookup-crc-full --variable-renaming --lookup-when-assert
--canonize --variable-simplification --variable-propagation
--propagation-lookup-crc-full --simplify
```

### Data Set 1 Result

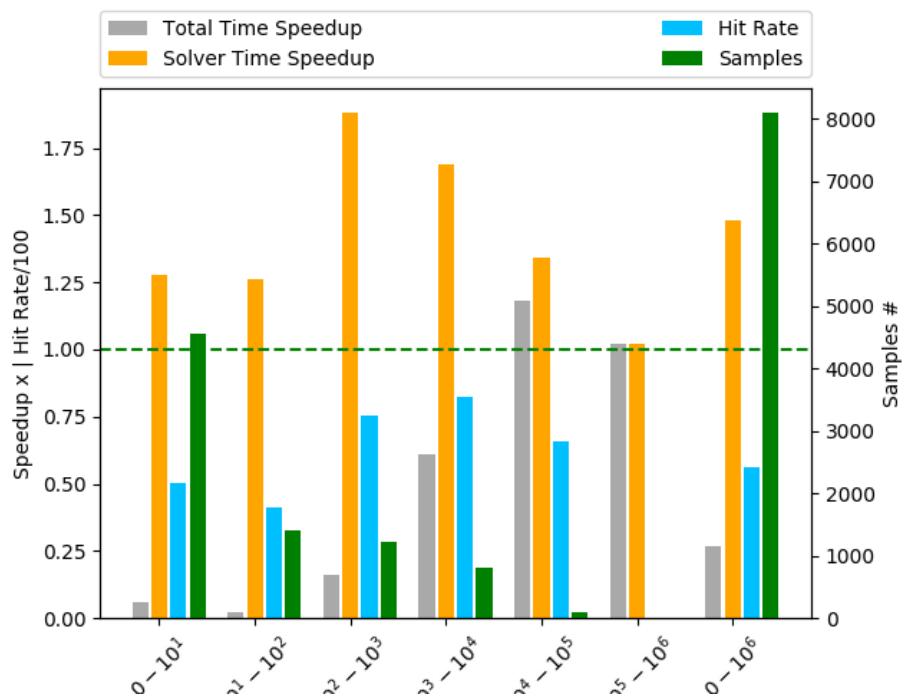
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	17475788	-66923	325782	2158.57	4.0	8096	5212	1914
1000	17172092	-66923	325782	12882.29	2418.0	1333	886	447
5000	16971664	-66923	325782	30579.57	14547.0	555	480	75
Solver Time (ms)								
0	-1872303	-67338	90859	-231.26	-1.0	8096	1021	4506
1000	-1578129	-67338	90859	-2248.05	-1736.0	702	119	583
5000	-730715	-67338	90859	-4970.85	-6525.0	147	41	106
Hit Rate (%)								
0	436880	0	100	56.31	88.0	8096	4841	0
Cache Hits (#)								
0	59916	0	208	7.72	1.0	8096	4841	0



## A.8. EXPERIMENT 8 RESULTS

145

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	2135.34	0.06	-0.40	1.28	50.55	4560
$10^1 - 10^2$	3016.86	0.02	-7.97	1.26	41.34	1401
$10^2 - 10^3$	1940.20	0.16	-158.67	1.88	75.27	1215
$10^3 - 10^4$	1921.90	0.61	-1197.51	1.69	82.53	816
$10^4 - 10^5$	-3989.08	1.18	-6666.22	1.34	65.64	103
$10^5 - 10^6$	-2725.00	1.02	-2732.00	1.02	0.00	1
$0 - 10^6$	2158.57	0.27	-231.26	1.48	56.31	8096



## A.9 Experiment 9 Results

### Setup

The following commands are used for this experiment:

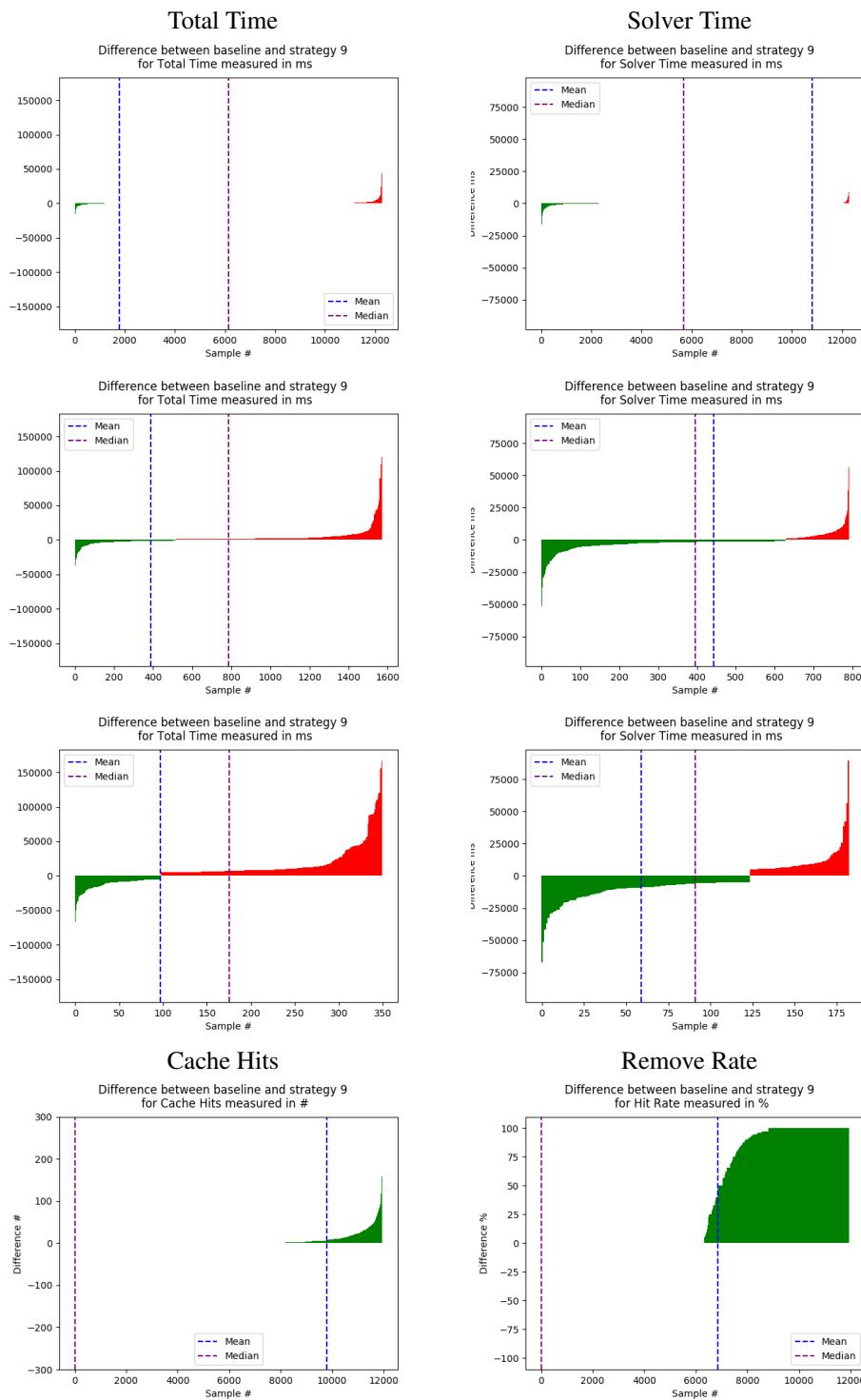
```
--cache --strat9
or
--cache --lookup --lookup-crc-full --variable-renaming --lookup-when-assert
--canonize --variable-simplification --variable-propagation
--propagation-lookup-crc-full --simplify-replaced --store-during-lookup
```

### Data Set 1 Results

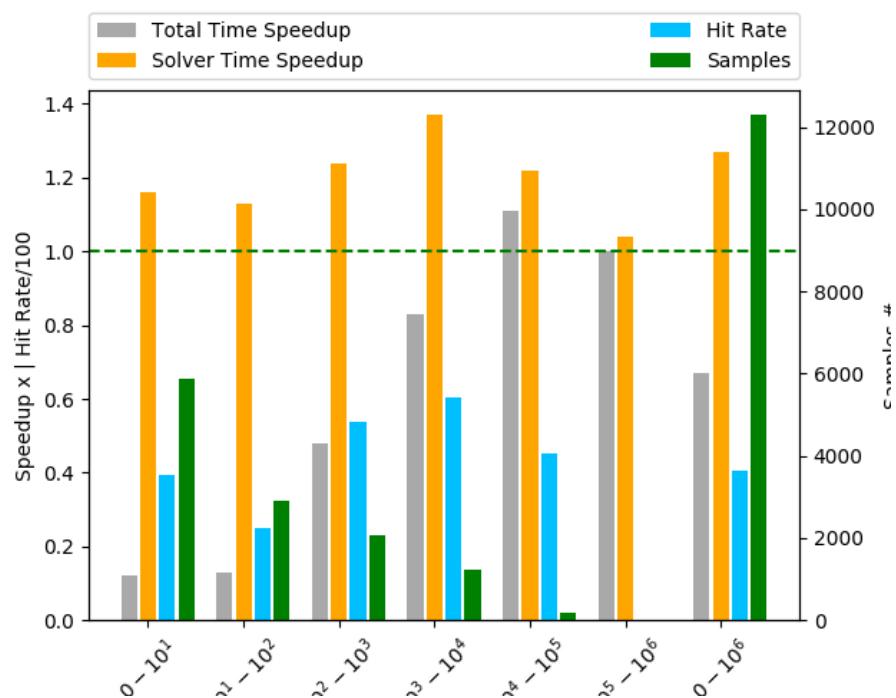
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	5082714	-67192	166507	413.26	9.0	12299	8717	2451
1000	4433843	-67192	166507	2818.72	1525.0	1573	1054	519
5000	3736967	-67192	166507	10677.05	6968.5	350	252	98
Solver Time (ms)								
0	-2007379	-67281	89131	-163.21	-1.0	12299	2290	6624
1000	-1603488	-67281	89131	-2024.61	-1657.5	792	164	628
5000	-747771	-67281	89131	-4086.18	-5970.0	183	59	124
Hit Rate (%)								
0	486808	0	100	40.7	0.0	12299	5655	0
Cache Hits (#)								
0	73432	0	273	6.14	0.0	12299	5655	0

## A.9. EXPERIMENT 9 RESULTS

147



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	440.06	0.12	-0.33	1.16	39.54	5884
$10^1 - 10^2$	429.58	0.13	-4.52	1.13	25.03	2922
$10^2 - 10^3$	444.89	0.48	-68.63	1.24	53.88	2060
$10^3 - 10^4$	636.79	0.83	-788.42	1.37	60.22	1247
$10^4 - 10^5$	-2562.82	1.11	-4647.27	1.22	45.04	184
$10^5 - 10^6$	-413.00	1.00	-6313.00	1.04	0.00	2
$0 - 10^6$	413.26	0.67	-163.21	1.27	40.70	12299



## A.10 Experiment 10 Results

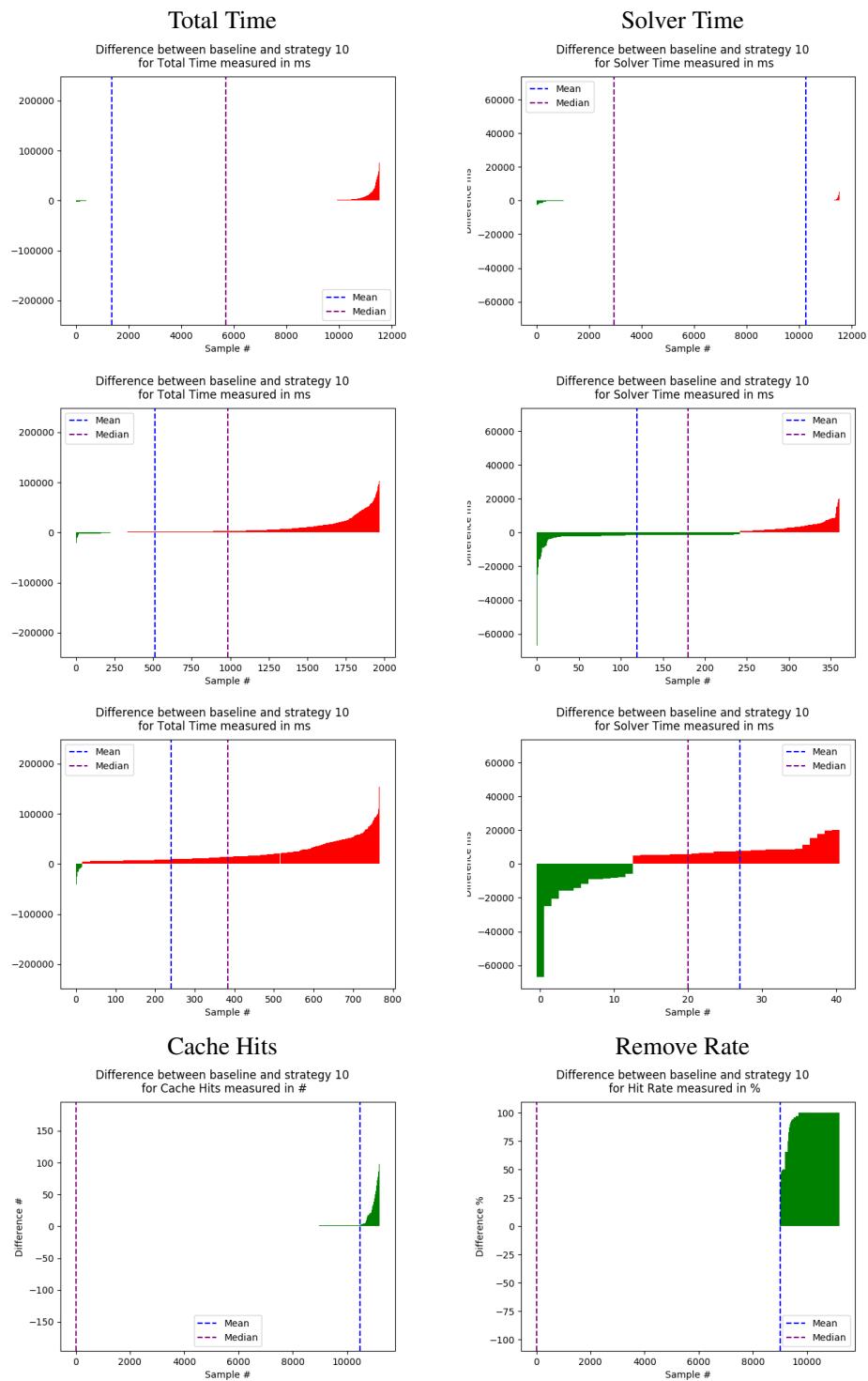
### Setup

The following commands are used for this experiment:

```
--cache --strat10
or
--cache --subset-lookup --subset-lookup-crc-full --lookup-when-assert
--use-rename-filter
```

### Data Set 1 Results

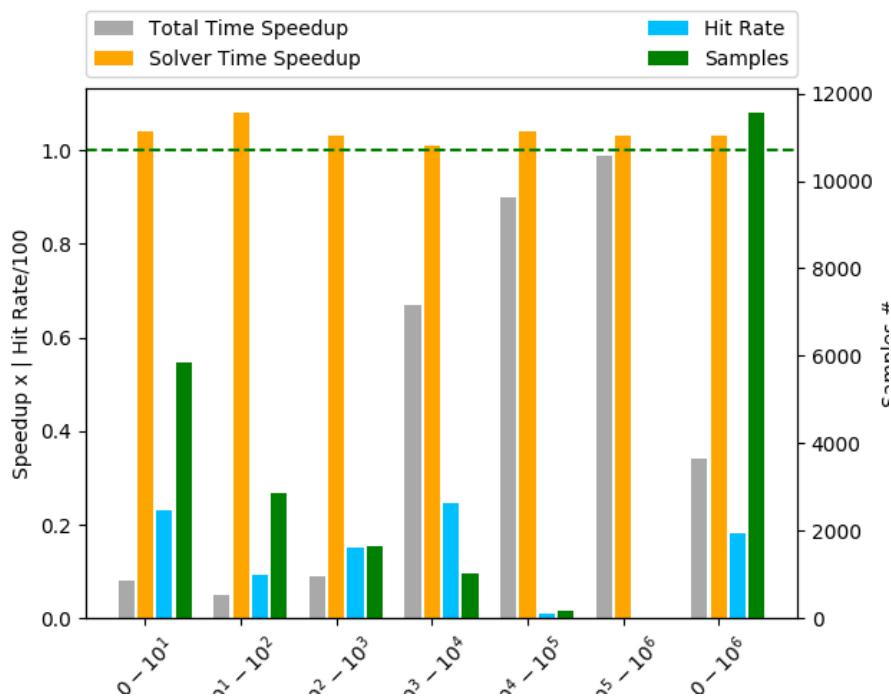
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	19440154	-67083	226258	1682.55	10.0	11554	8690	1649
1000	18780668	-67083	226258	9533.33	3082.0	1970	1747	223
5000	16843622	-67083	226258	21931.8	14035.5	768	752	16
Solver Time (ms)								
0	-220500	-67087	19989	-19.08	0.0	11554	2951	4822
1000	-163291	-67087	19989	-452.33	-1480.0	361	119	242
5000	24524	-67087	19989	598.15	5898.0	41	28	13
Hit Rate (%)								
0	204589	0	100	18.24	0.0	11554	2223	0
Cache Hits (#)								
0	25017	0	178	2.23	0.0	11554	2252	0



## A.10. EXPERIMENT 10 RESULTS

151

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	625.41	0.08	-0.10	1.04	23.20	5850
$10^1 - 10^2$	1792.46	0.05	-2.76	1.08	9.25	2859
$10^2 - 10^3$	5150.13	0.09	-9.82	1.03	15.14	1647
$10^3 - 10^4$	1627.89	0.67	-20.48	1.01	24.46	1027
$10^4 - 10^5$	3022.02	0.90	-918.74	1.04	1.12	165
$10^5 - 10^6$	683.00	0.99	-3871.17	1.03	0.00	6
$0 - 10^6$	1682.55	0.34	-19.08	1.03	18.24	11554



## A.11 Experiment 11 Results

### Setup

The following commands are used for this experiment:

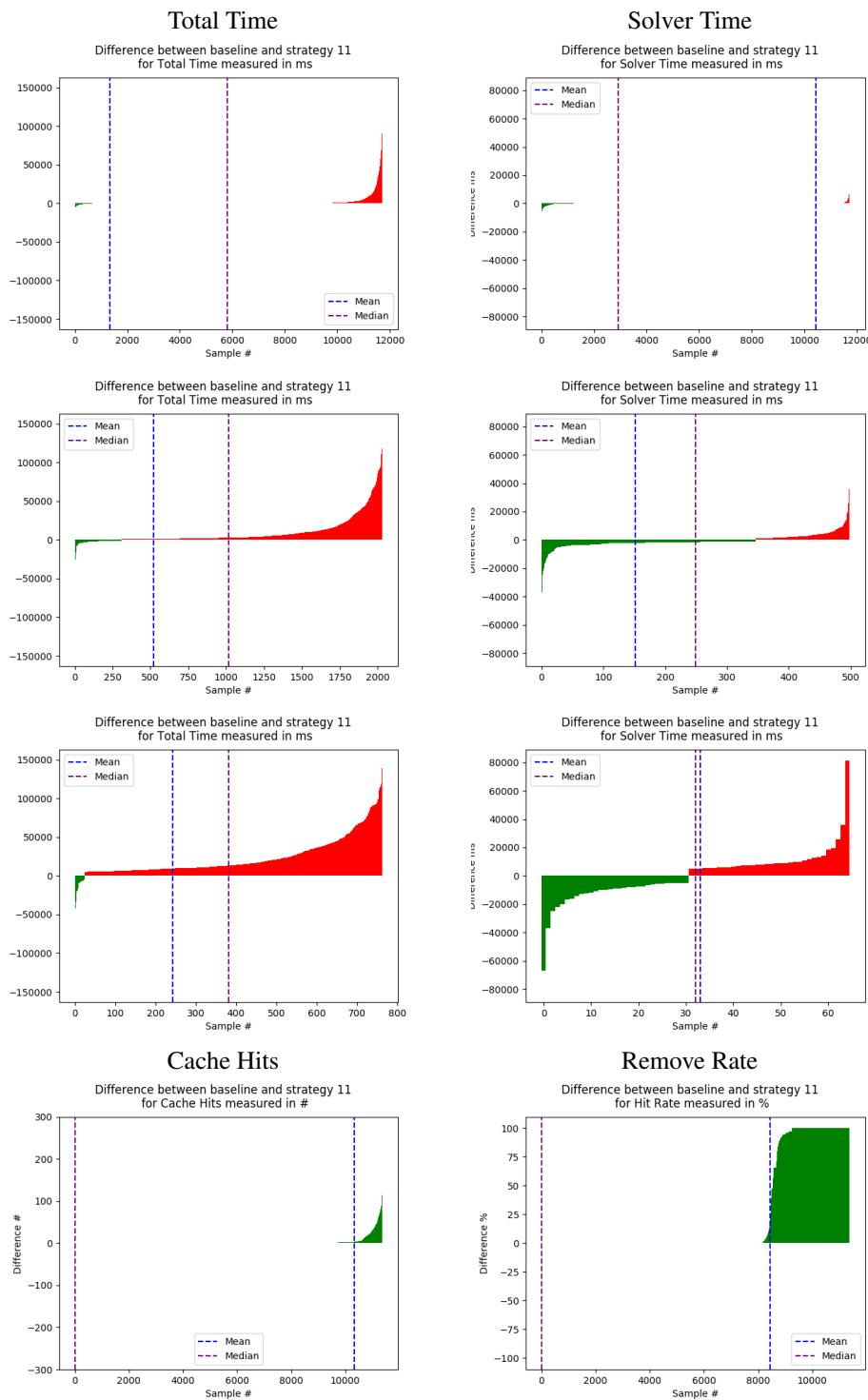
```
--cache --strat11
or
--cache --subset-lookup --subset-lookup-crc-full --lookup-when-assert
--use-rename-filter --use-shape-filter
```

### Data Set 1 Results

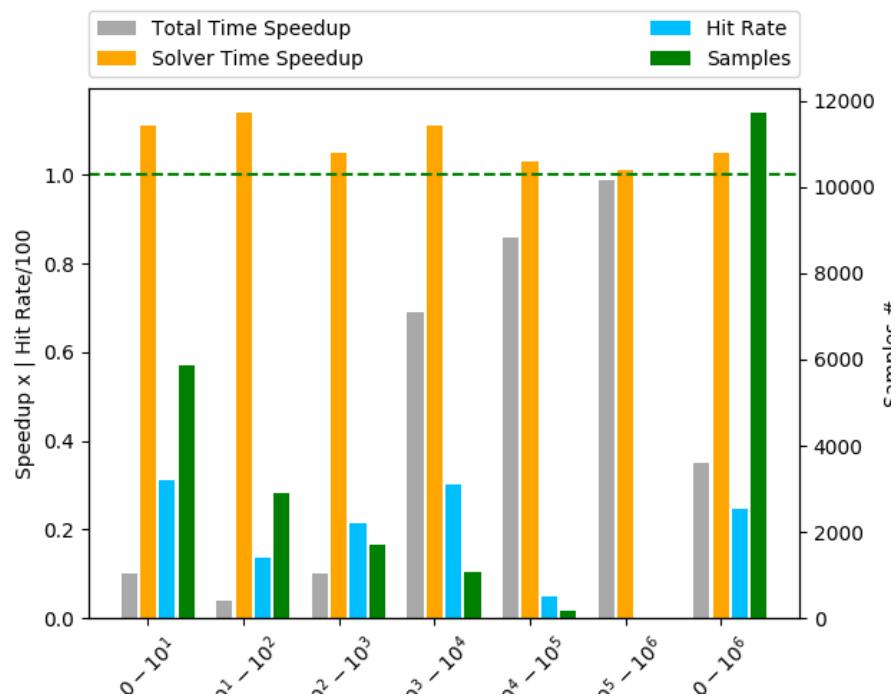
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	19904135	-66847	148543	1698.45	7.0	11719	8348	2160
1000	19258416	-66847	148543	9468.25	2795.0	2034	1724	310
5000	17651213	-66847	148543	23103.68	13267.0	764	739	25
Solver Time (ms)								
0	-467651	-66852	81107	-39.91	0.0	11719	2923	5215
1000	-384584	-66852	81107	-770.71	-1510.0	499	152	347
5000	14246	-66852	81107	219.17	5185.0	65	34	31
Hit Rate (%)								
0	280544	0	100	24.65	0.0	11719	3216	0
Cache Hits (#)								
0	34750	0	273	3.05	0.0	11719	3260	0

## A.11. EXPERIMENT 11 RESULTS

153



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	562.63	0.10	-0.23	1.11	31.04	5875
$10^1 - 10^2$	1862.88	0.04	-4.92	1.14	13.52	2897
$10^2 - 10^3$	5201.57	0.10	-16.33	1.05	21.38	1698
$10^3 - 10^4$	1462.48	0.69	-267.70	1.11	30.31	1067
$10^4 - 10^5$	4540.16	0.86	-727.98	1.03	5.04	176
$10^5 - 10^6$	1683.83	0.99	-1759.67	1.01	0.00	6
$0 - 10^6$	1698.45	0.35	-39.91	1.05	24.65	11719



## A.12 Experiment 12 Results

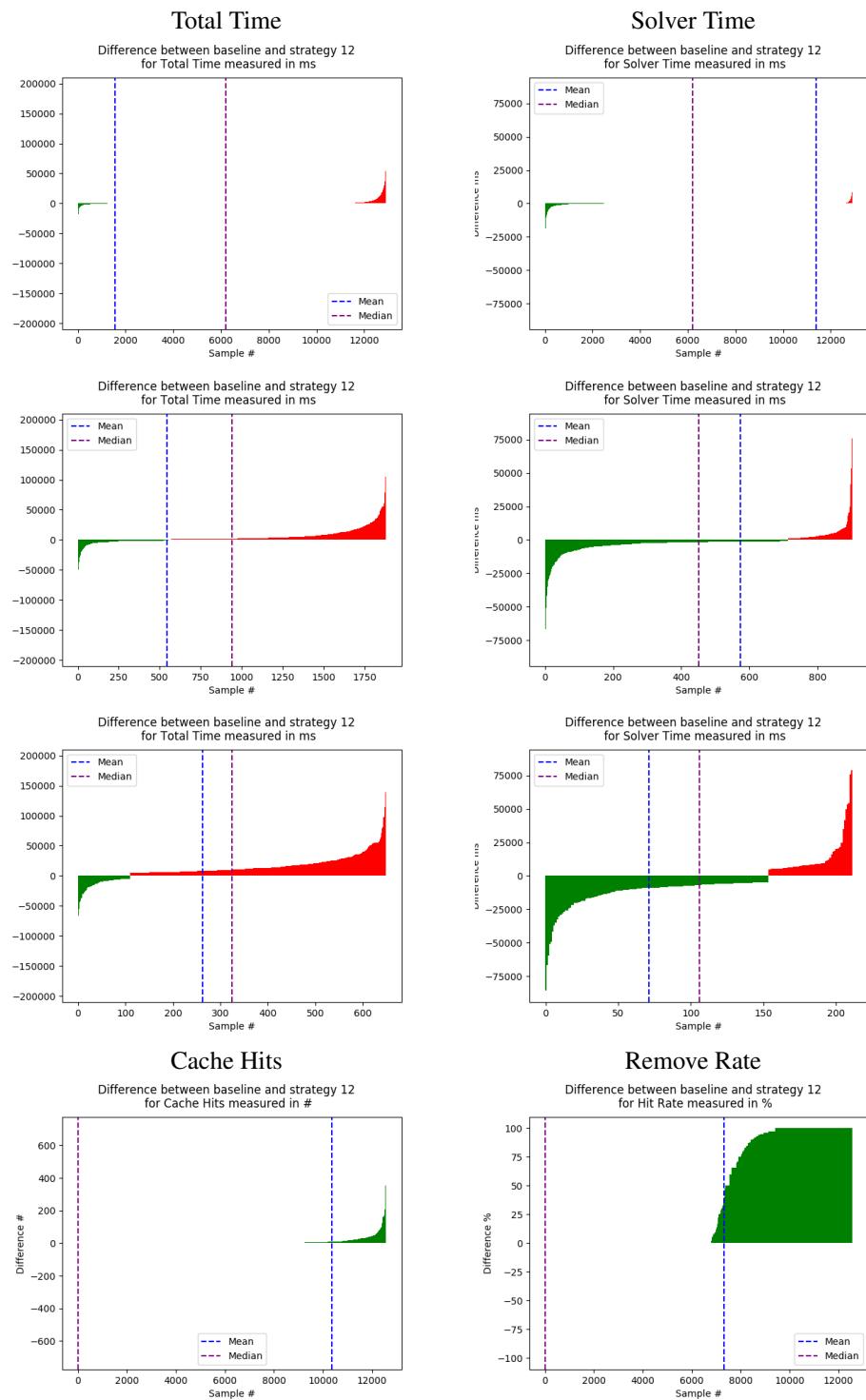
### Setup

The following commands are used for this experiment:

```
--cache --strat11
or
--cache --subset-lookup --subset-lookup-crc-full --lookup-when-assert
--use-rename-filter --use-shape-filter --use-range-filter
```

### Data Set 1 Results

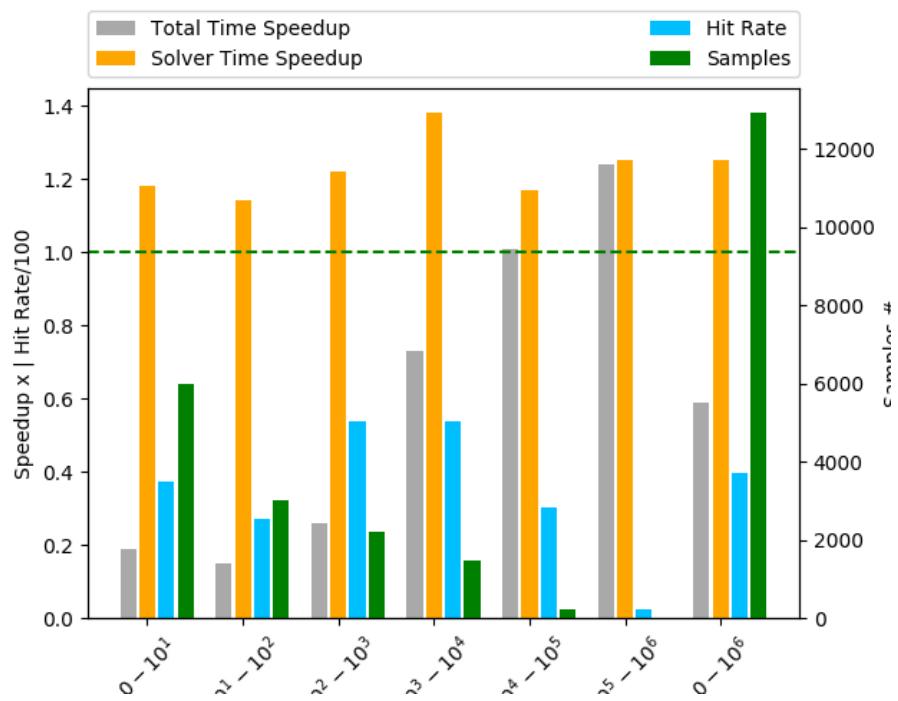
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	9744889	-84161	191275	754.42	4.0	12917	8134	3289
1000	9245300	-84161	191275	4907.27	2010.5	1884	1332	552
5000	8411136	-84161	191275	12940.21	10030.5	650	540	110
Solver Time (ms)								
0	-2356453	-85897	79196	-182.43	-1.0	12917	2711	6726
1000	-1986156	-85897	79196	-2197.08	-1615.0	904	190	714
5000	-1115576	-85897	79196	-5262.15	-7023.0	212	58	154
Hit Rate (%)								
0	496923	0	100	39.5	0.0	12917	5794	0
Cache Hits (#)								
0	110472	0	705	8.78	0.0	12917	5800	0



## A.12. EXPERIMENT 12 RESULTS

157

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	397.98	0.19	-0.36	1.18	37.34	5976
$10^1 - 10^2$	526.95	0.15	-4.68	1.14	27.03	3020
$10^2 - 10^3$	1869.03	0.26	-65.51	1.22	53.61	2219
$10^3 - 10^4$	1237.40	0.73	-802.89	1.38	53.62	1476
$10^4 - 10^5$	-246.32	1.01	-3912.56	1.17	30.30	220
$10^5 - 10^6$	-24068.17	1.24	-24825.17	1.25	2.50	6
$0 - 10^6$	754.42	0.59	-182.43	1.25	39.50	12917



## A.13 Experiment 13 Results

### Setup

The following commands are used for this experiment:

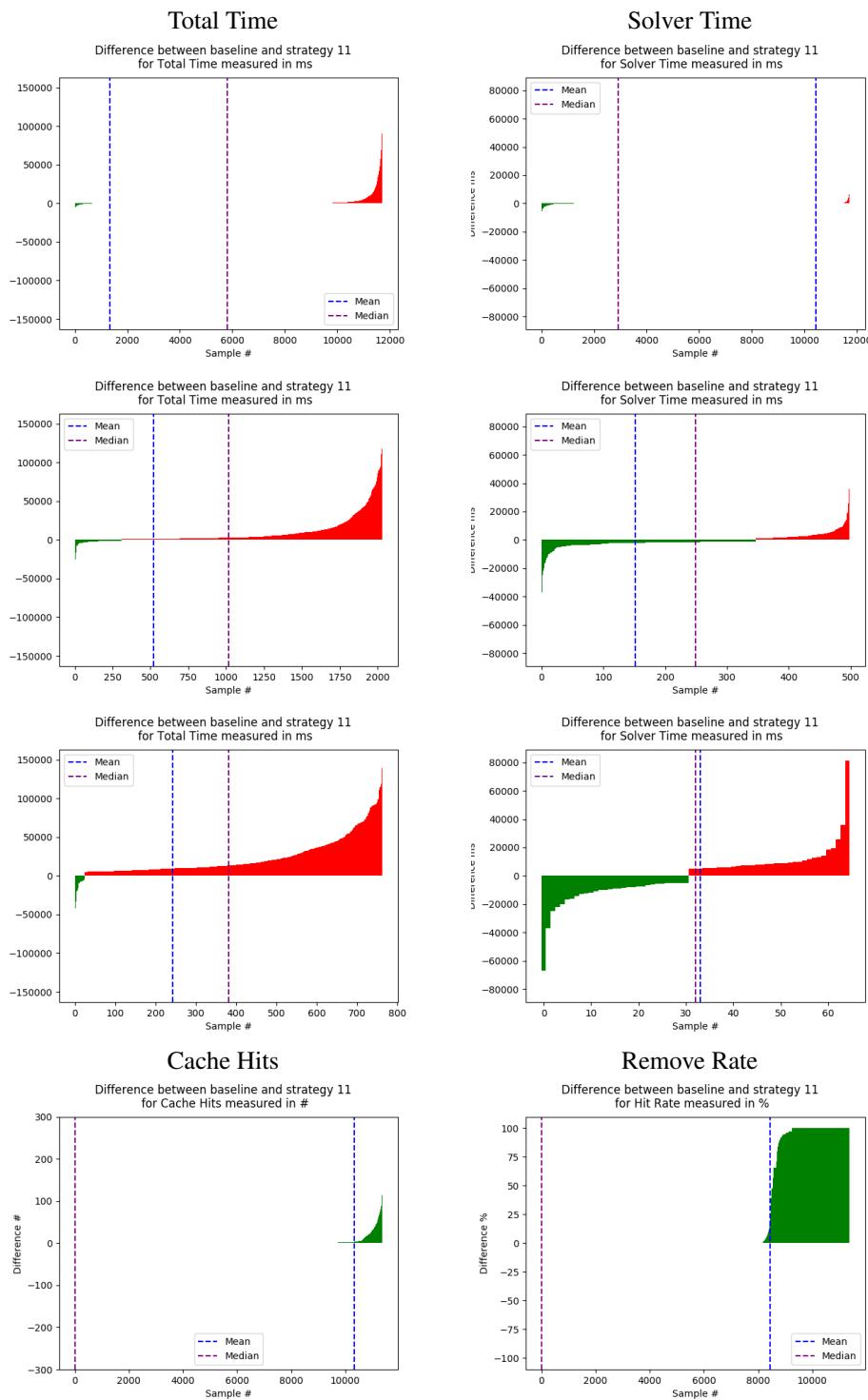
```
--cache --strat13
or
--cache --subset-lookup --subset-lookup-crc-full --lookup-when-assert
--use-rename-filter --use-shape-filter --use-range-filter
--use-forward-and-backward-filters
```

### Data Set 1 Results

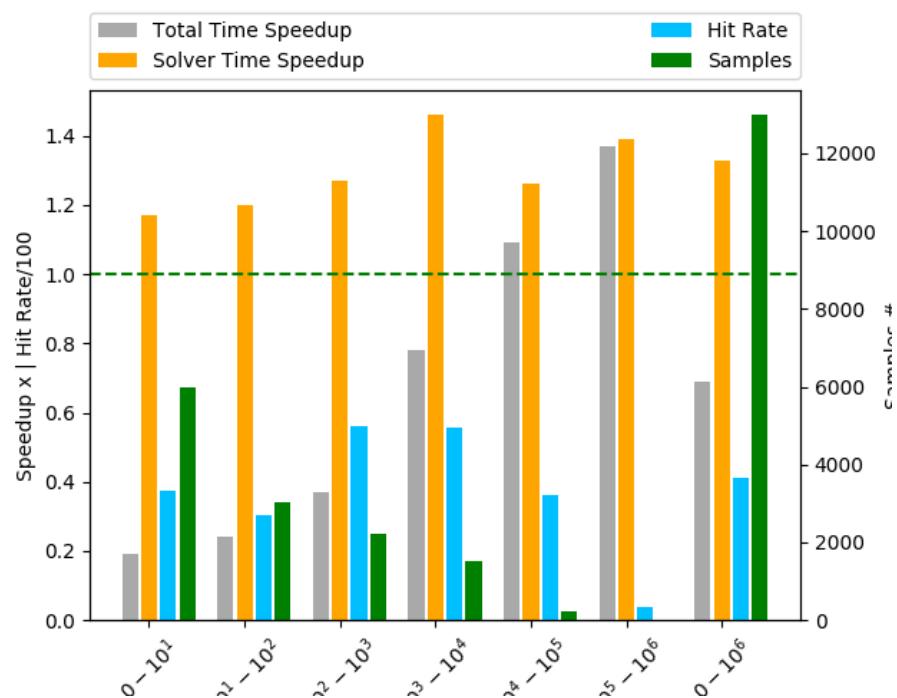
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	6370267	-88488	191486	490.59	4.0	12985	8163	3331
1000	5833740	-88488	191486	3165.35	1727.0	1843	1251	592
5000	5083339	-88488	191486	9159.17	8598.0	555	441	114
Solver Time (ms)								
0	-3047777	-90278	73457	-234.72	-1.0	12985	2548	6967
1000	-2605438	-90278	73457	-2816.69	-1746.0	925	160	765
5000	-1585879	-90278	73457	-7208.54	-7297.0	220	48	172
Hit Rate (%)								
0	521002	0	100	41.2	0.0	12985	5991	0
Cache Hits (#)								
0	113344	0	705	8.96	0.0	12985	5995	0

## A.13. EXPERIMENT 13 RESULTS

159



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	397.23	0.19	-0.35	1.17	37.44	5976
$10^1 - 10^2$	289.88	0.24	-6.47	1.20	30.54	3020
$10^2 - 10^3$	1069.82	0.37	-76.61	1.27	55.90	2230
$10^3 - 10^4$	937.17	0.78	-919.69	1.46	55.62	1521
$10^4 - 10^5$	-2223.09	1.09	-5499.85	1.26	36.12	233
$10^5 - 10^6$	-34435.40	1.37	-34995.80	1.39	3.80	5
$0 - 10^6$	490.59	0.69	-234.72	1.33	41.20	12985



## A.14 Experiment 14 Results

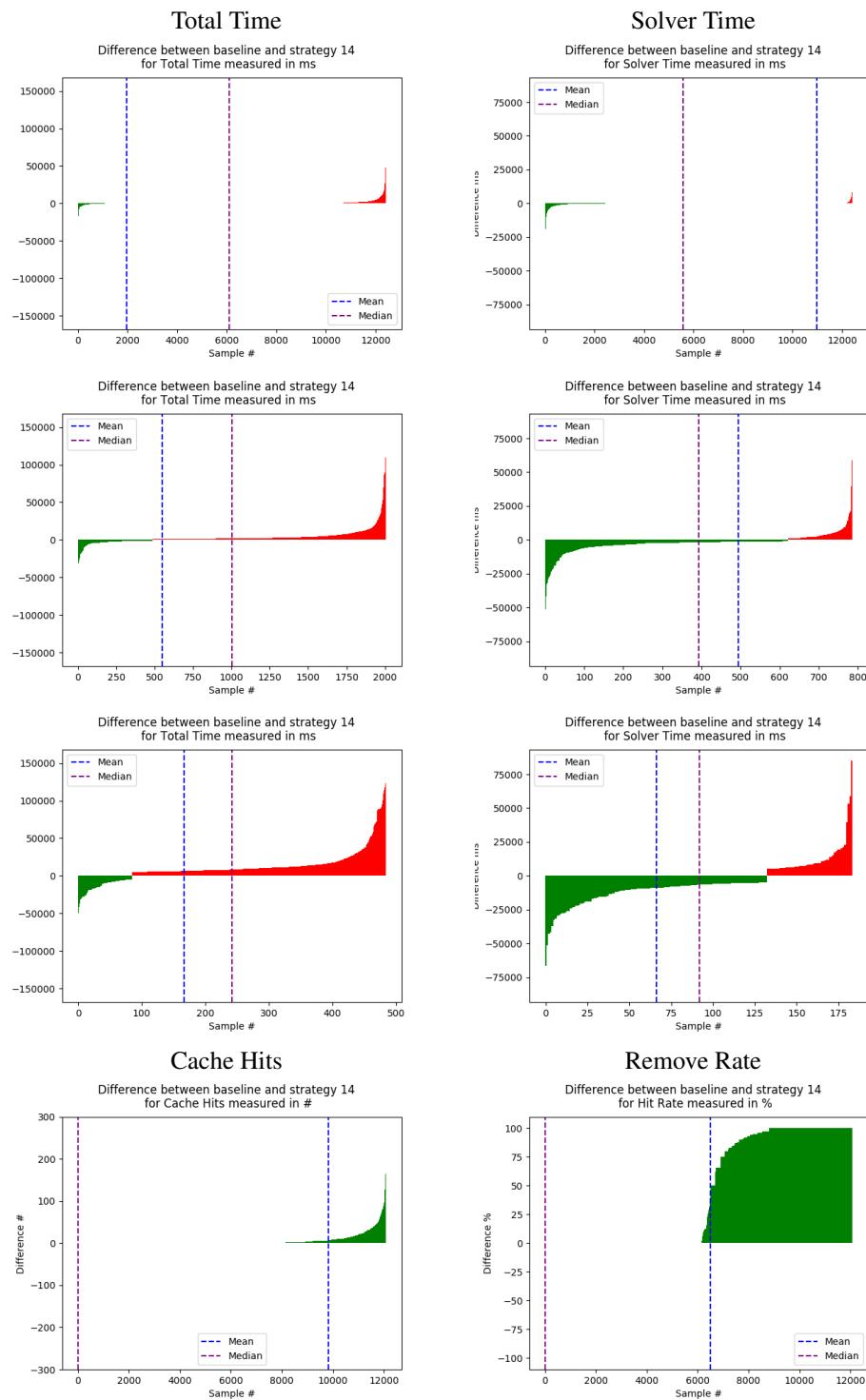
### Setup

The following commands are used for this experiment:

```
--cache --strat14
or
--cache --subset-lookup --subset-lookup-crc-full --lookup-when-assert
--use-rename-filter --use-shape-filter --use-range-filter
--use-forward-and-backward-filters
--canonize --variable-simplification --variable-propagation
--simplify-replaced
```

### Data Set 1 Results

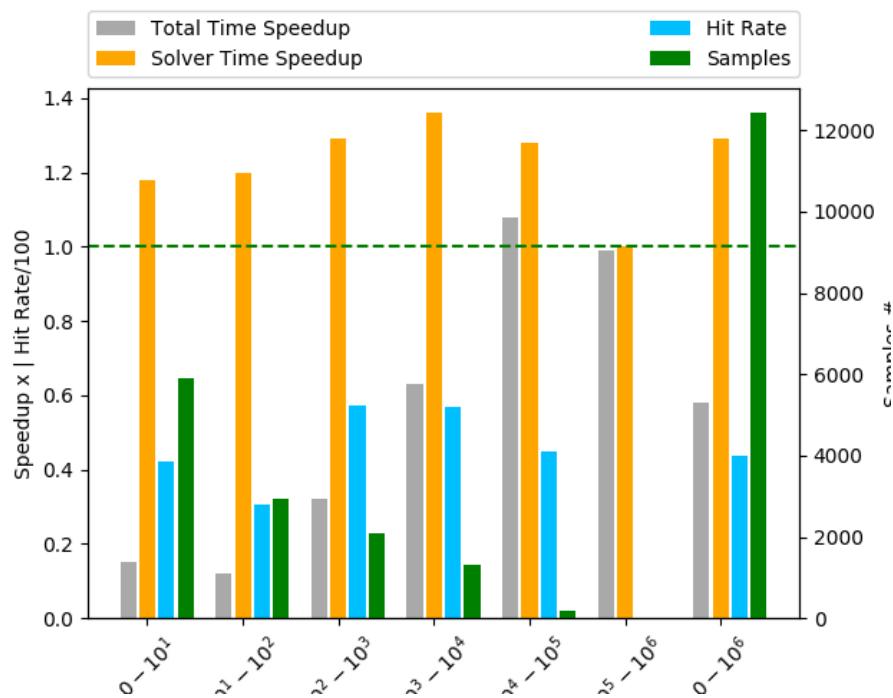
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	7865735	-66598	152903	633.01	11.0	12426	8891	2404
1000	7122596	-66598	152903	3547.11	1956.0	2008	1521	487
5000	5469789	-66598	152903	11277.92	8381.0	485	400	85
Solver Time (ms)								
0	-2235713	-66657	84927	-179.92	-1.0	12426	2105	6854
1000	-1803382	-66657	84927	-2291.46	-1703.0	787	164	623
5000	-988389	-66657	84927	-5371.68	-6525.0	184	51	133
Hit Rate (%)								
0	528218	0	100	43.7	0.0	12426	5927	0
Cache Hits (#)								
0	75776	0	273	6.27	0.0	12426	5927	0



## A.14. EXPERIMENT 14 RESULTS

163

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	448.31	0.15	-0.37	1.18	42.35	5905
$10^1 - 10^2$	482.38	0.12	-6.59	1.20	30.53	2934
$10^2 - 10^3$	881.01	0.32	-79.58	1.29	57.42	2093
$10^3 - 10^4$	1761.15	0.63	-752.58	1.36	56.87	1312
$10^4 - 10^5$	-1969.48	1.08	-5889.58	1.28	44.97	180
$10^5 - 10^6$	1522.50	0.99	-64.50	1.00	0.00	2
$0 - 10^6$	633.01	0.58	-179.92	1.29	43.70	12426



## A.15 Experiment 15 Results

### Setup

The following commands are used for this experiment:

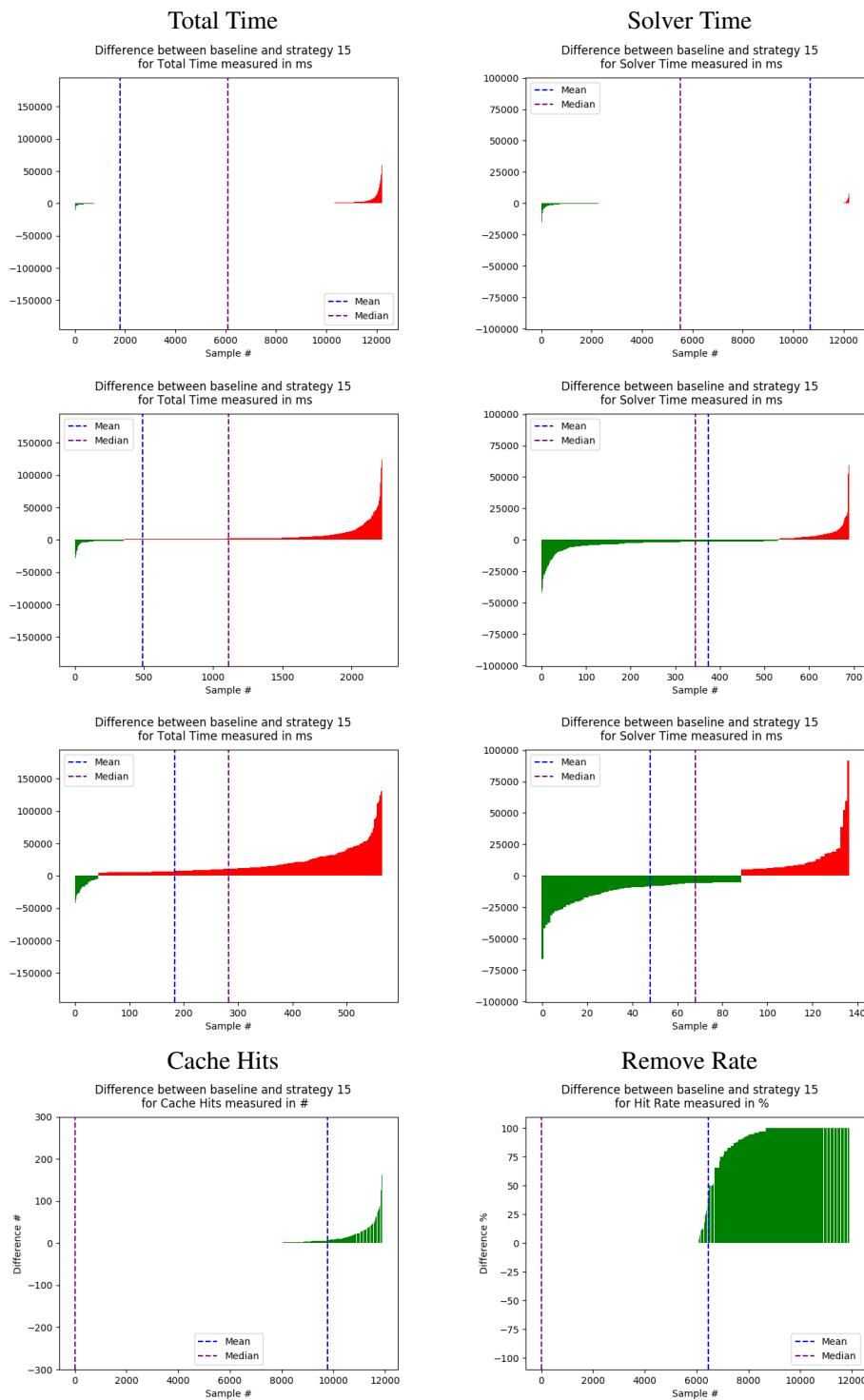
```
--cache --strat15
or
--cache --subset-lookup --subset-lookup-crc-full --lookup-when-assert
--use-rename-filter --use-shape-filter --use-range-filter
--use-forward-and-backward-filters
--canonize --variable-simplification --variable-propagation
--simplify-replaced --store-during-subset-lookup
```

### Data Set 1 Results

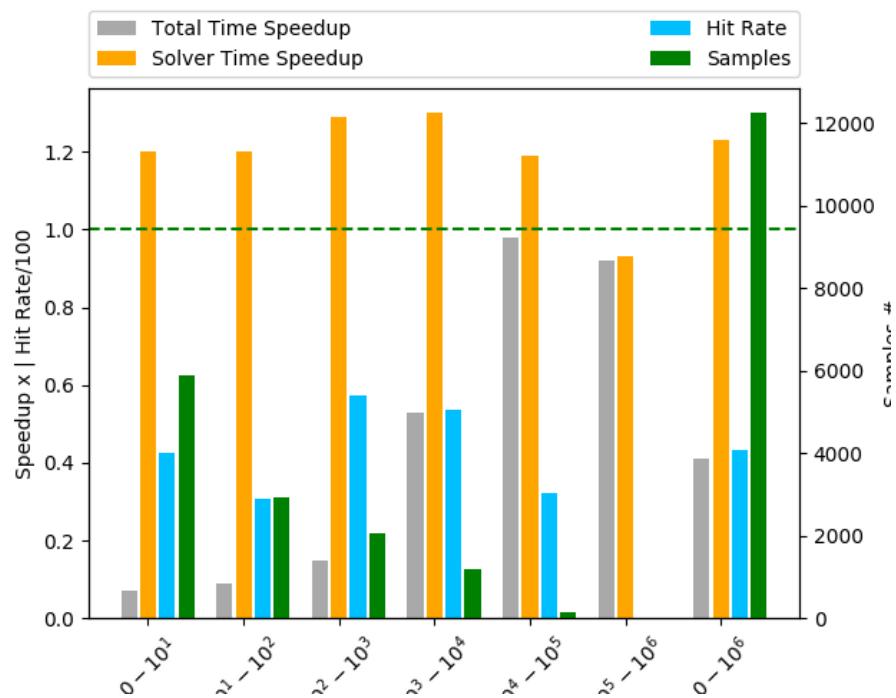
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	12974323	-65968	177403	1059.82	15.0	12242	9405	1830
1000	12117015	-65968	177403	5448.3	2032.0	2224	1867	357
5000	9924625	-65968	177403	17534.67	10947.5	566	523	43
Solver Time (ms)								
0	-1542117	-66069	91387	-125.97	-1.0	12242	2060	6709
1000	-1150204	-66069	91387	-1664.55	-1600.0	691	160	531
5000	-466827	-66069	91387	-3407.5	-5793.0	137	48	89
Hit Rate (%)								
0	514264	0	100	43.2	0.0	12242	5831	0
Cache Hits (#)								
0	74132	0	273	6.23	0.0	12242	5868	0

## A.15. EXPERIMENT 15 RESULTS

165



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	488.35	0.07	-0.40	1.20	42.56	5890
$10^1 - 10^2$	671.47	0.09	-6.60	1.20	30.65	2933
$10^2 - 10^3$	2384.14	0.15	-80.71	1.29	57.33	2076
$10^3 - 10^4$	2577.19	0.53	-641.32	1.30	53.75	1199
$10^4 - 10^5$	446.18	0.98	-4260.15	1.19	32.25	142
$10^5 - 10^6$	12817.50	0.92	10530.50	0.93	0.00	2
$0 - 10^6$	1059.82	0.41	-125.97	1.23	43.20	12242



## A.16 Experiment 16 Results

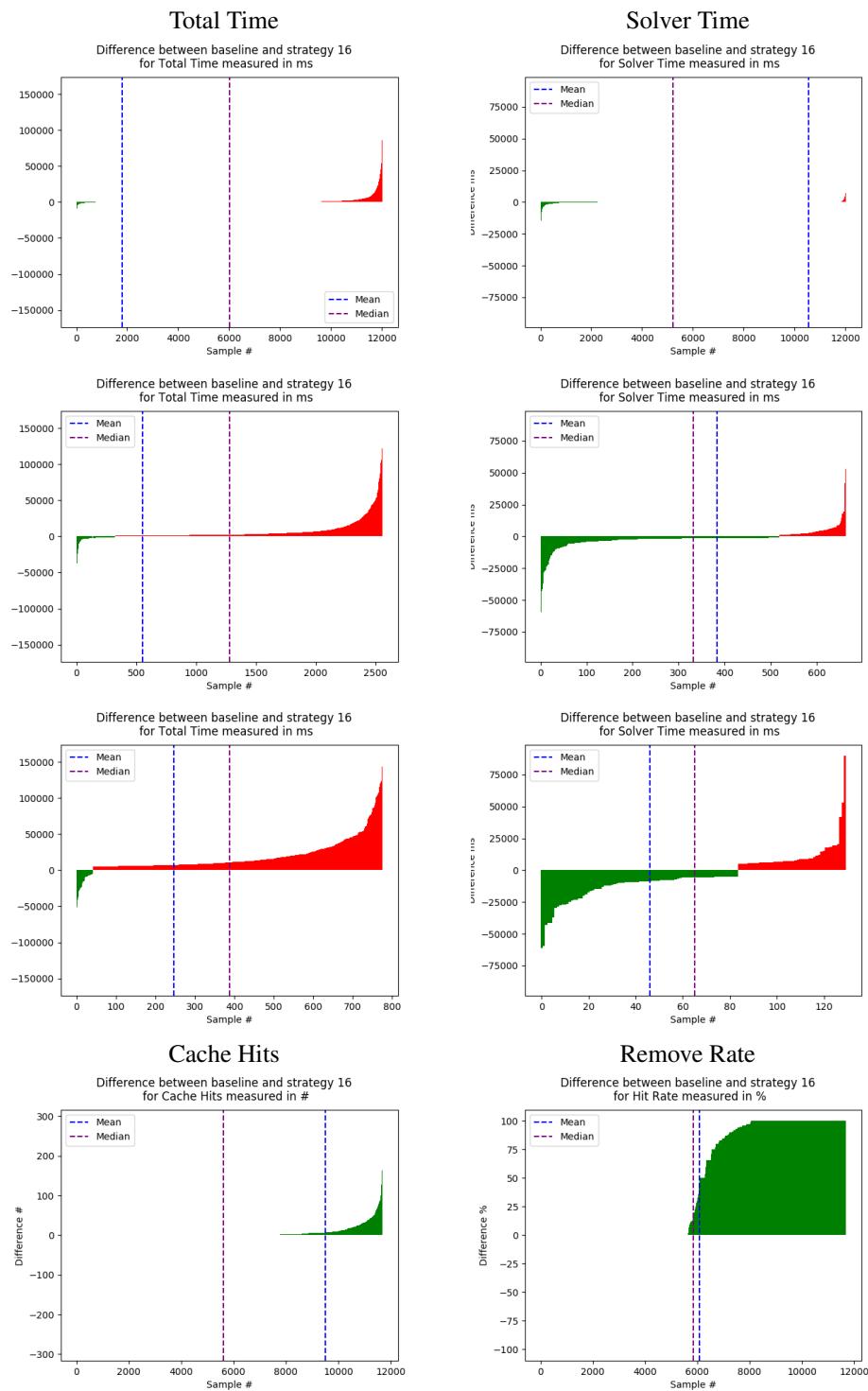
### Setup

The following commands are used for this experiment:

```
--cache --strat16
or
--cache --subset-lookup --subset-lookup-crc-full --lookup-when-assert
--use-rename-filter --use-shape-filter --use-range-filter
--use-forward-and-backward-filters
--canonize --variable-simplification --variable-propagation
--simplify-replaced --store-during-subset-lookup
--split-assumptions-and-assert
```

### Data Set 1 Results

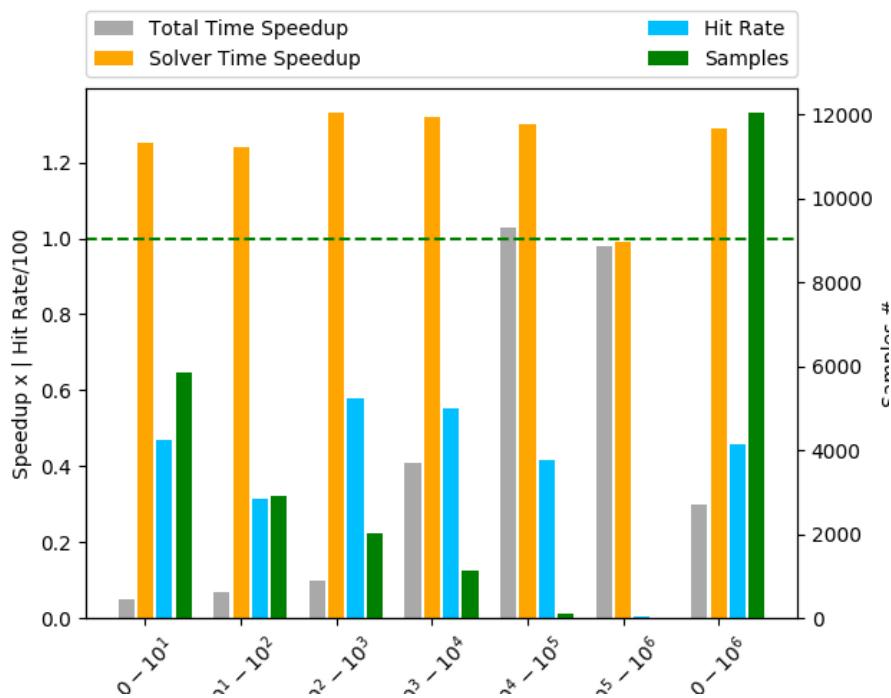
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	18235736	-61237	158254	1514.6	21.0	12040	9305	1745
1000	17334253	-61237	158254	6771.19	2417.5	2560	2236	324
5000	14476063	-61237	158254	18630.71	10774.0	777	735	42
Solver Time (ms)								
0	-1659428	-61243	89680	-137.83	-1.0	12040	1846	6822
1000	-1257724	-61243	89680	-1891.31	-1606.0	665	145	520
5000	-588367	-61243	89680	-4525.9	-5728.5	130	46	84
Hit Rate (%)								
0	534183	0	100	45.65	17.0	12040	6050	0
Cache Hits (#)								
0	77102	0	288	6.59	1.0	12040	6086	0



## A.16. EXPERIMENT 16 RESULTS

169

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	684.22	0.05	-0.47	1.25	46.74	5845
$10^1 - 10^2$	902.03	0.07	-7.57	1.24	31.38	2900
$10^2 - 10^3$	3488.70	0.10	-87.73	1.33	58.07	2037
$10^3 - 10^4$	4055.01	0.41	-646.58	1.32	55.06	1135
$10^4 - 10^5$	-791.08	1.03	-5992.23	1.30	41.59	121
$10^5 - 10^6$	3703.50	0.98	1460.00	0.99	0.50	2
$0 - 10^6$	1514.60	0.30	-137.83	1.29	45.65	12040



## A.17 Experiment 17 Results

### Setup

The following commands are used for this experiment:

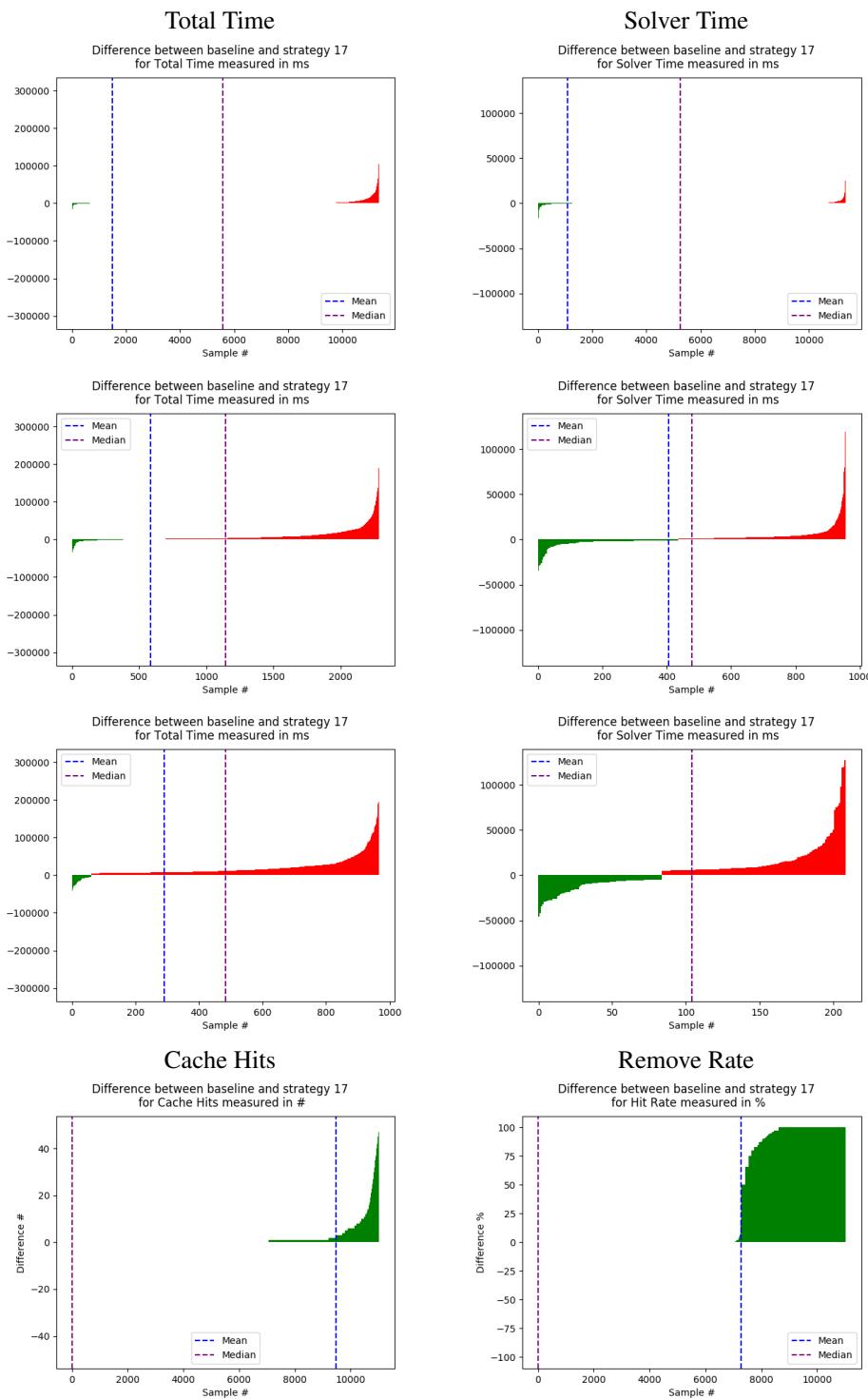
```
--cache --strat17
or
--cache --unsat-core-lookup --unsat-core-lookup-crc-full --
-lookup-when-assert
--use-rename-filter --use-shape-filter --use-range-filter
--use-forward-and-backward-filters
```

### Data Set 1 Results

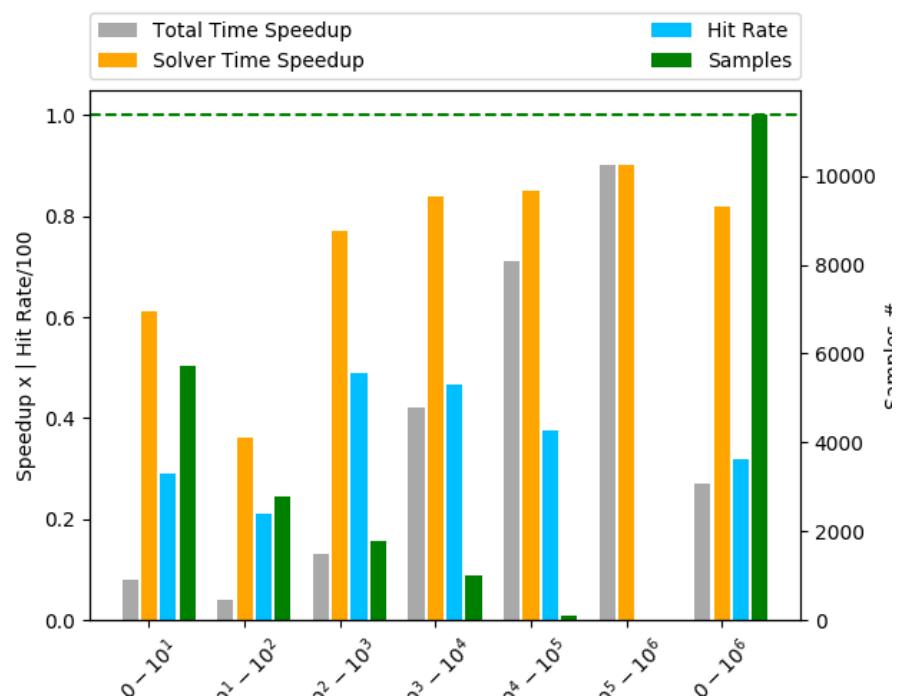
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	20829614	-45759	305025	1831.98	9.0	11370	7840	2203
1000	20399839	-45759	305025	8916.01	3371.0	2288	1906	382
5000	18616969	-45759	305025	19232.41	11513.5	968	907	61
Solver Time (ms)								
0	1399526	-45758	127035	123.09	0.0	11370	5246	3434
1000	1402599	-45758	127035	1465.62	1108.0	957	520	437
5000	1246481	-45758	127035	5964.02	5968.0	209	125	84
Hit Rate (%)								
0	352674	0	100	31.97	0.0	11370	3969	0
Cache Hits (#)								
0	22459	0	49	2.04	0.0	11370	3970	0

## A.17. EXPERIMENT 17 RESULTS

171



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	870.83	0.08	1.53	0.61	29.11	5714
$10^1 - 10^2$	1697.49	0.04	66.49	0.36	21.12	2769
$10^2 - 10^3$	3197.90	0.13	107.17	0.77	48.97	1772
$10^3 - 10^4$	4315.03	0.42	506.96	0.84	46.63	1004
$10^4 - 10^5$	10310.27	0.71	4341.38	0.85	37.58	108
$10^5 - 10^6$	13616.00	0.90	12966.00	0.90	0.00	3
$0 - 10^6$	1831.98	0.27	123.09	0.82	31.97	11370



## A.18 Experiment 18 Results

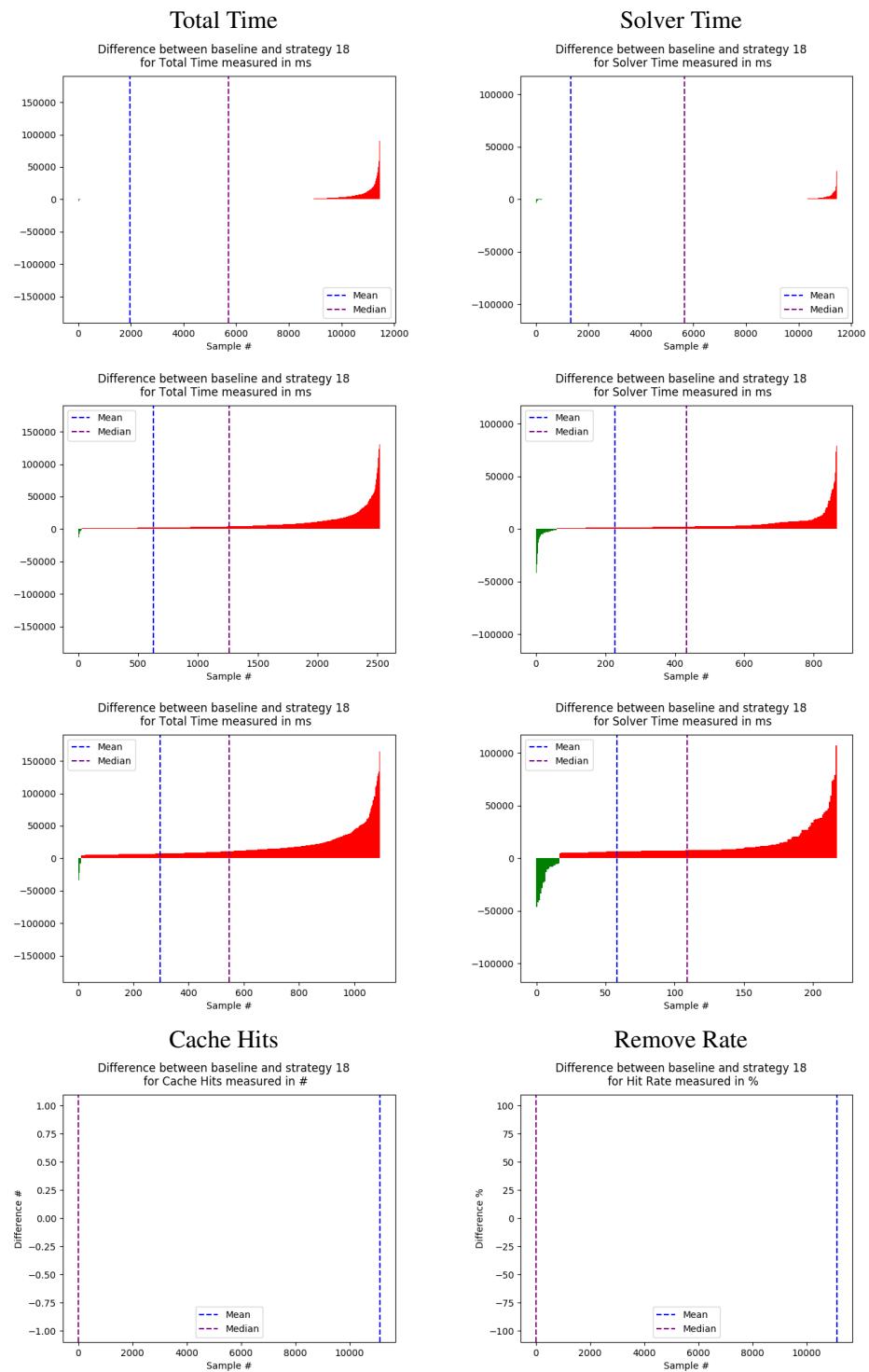
### Setup

The following commands are used for this experiment:

```
--cache --strat18
or
--cache --unsat-core-lookup --unsat-core-lookup-crc-full --
-lookup-when-assert
--use-rename-filter --use-shape-filter --use-range-filter
--use-forward-and-backward-filters --canonize
--variable-simplification --variable-propagation
--simplify-replaced
```

### Data Set 1 Results

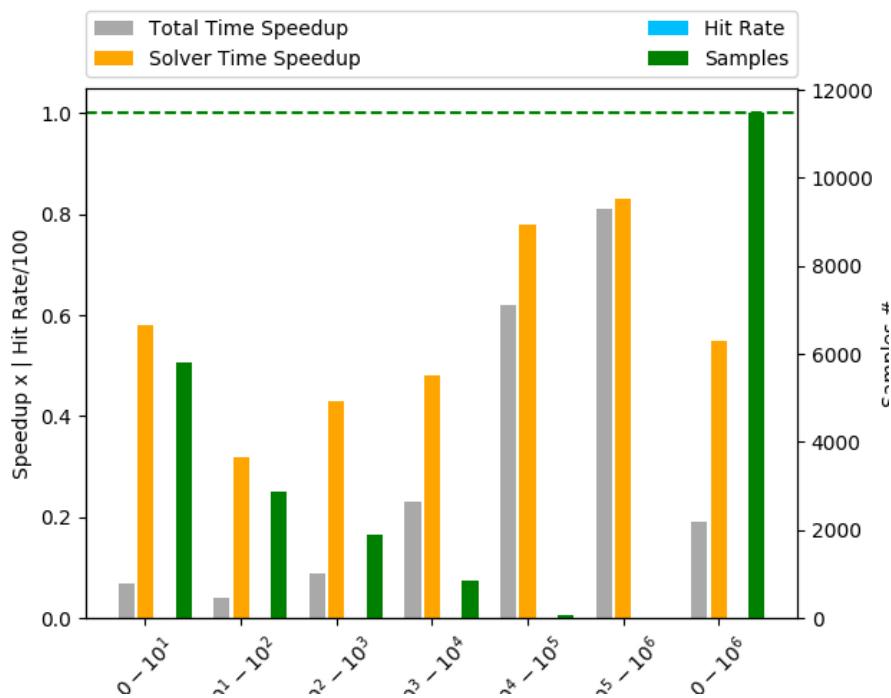
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	24288567	-46035	173432	2117.02	32.0	11473	10098	459
1000	23478423	-46035	173432	9298.39	4117.0	2525	2495	30
5000	19802313	-46035	173432	18084.3	10906.0	1095	1083	12
Solver Time (ms)								
0	4067307	-46036	107047	354.51	1.0	11473	7021	1743
1000	3588088	-46036	107047	4133.74	2390.5	868	807	61
5000	2371101	-46036	107047	10876.61	7611.5	218	201	17
Hit Rate (%)								
0	800	0	100	0.07	0.0	11473	8	0
Cache Hits (#)								
0	8	0	1	0.0	0.0	11473	8	0



## A.18. EXPERIMENT 18 RESULTS

175

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	676.32	0.07	1.73	0.58	0.15	5801
$10^1 - 10^2$	1370.40	0.04	81.57	0.32	0.00	2862
$10^2 - 10^3$	4030.51	0.09	462.56	0.43	0.00	1888
$10^3 - 10^4$	9109.86	0.23	2847.01	0.48	0.00	843
$10^4 - 10^5$	14403.05	0.62	6713.95	0.78	0.00	78
$10^5 - 10^6$	30525.00	0.81	26800.00	0.83	0.00	1
$0 - 10^6$	2117.02	0.19	354.51	0.55	0.07	11473



## A.19 Experiment 19 Results

### Setup

The following commands are used for this experiment:

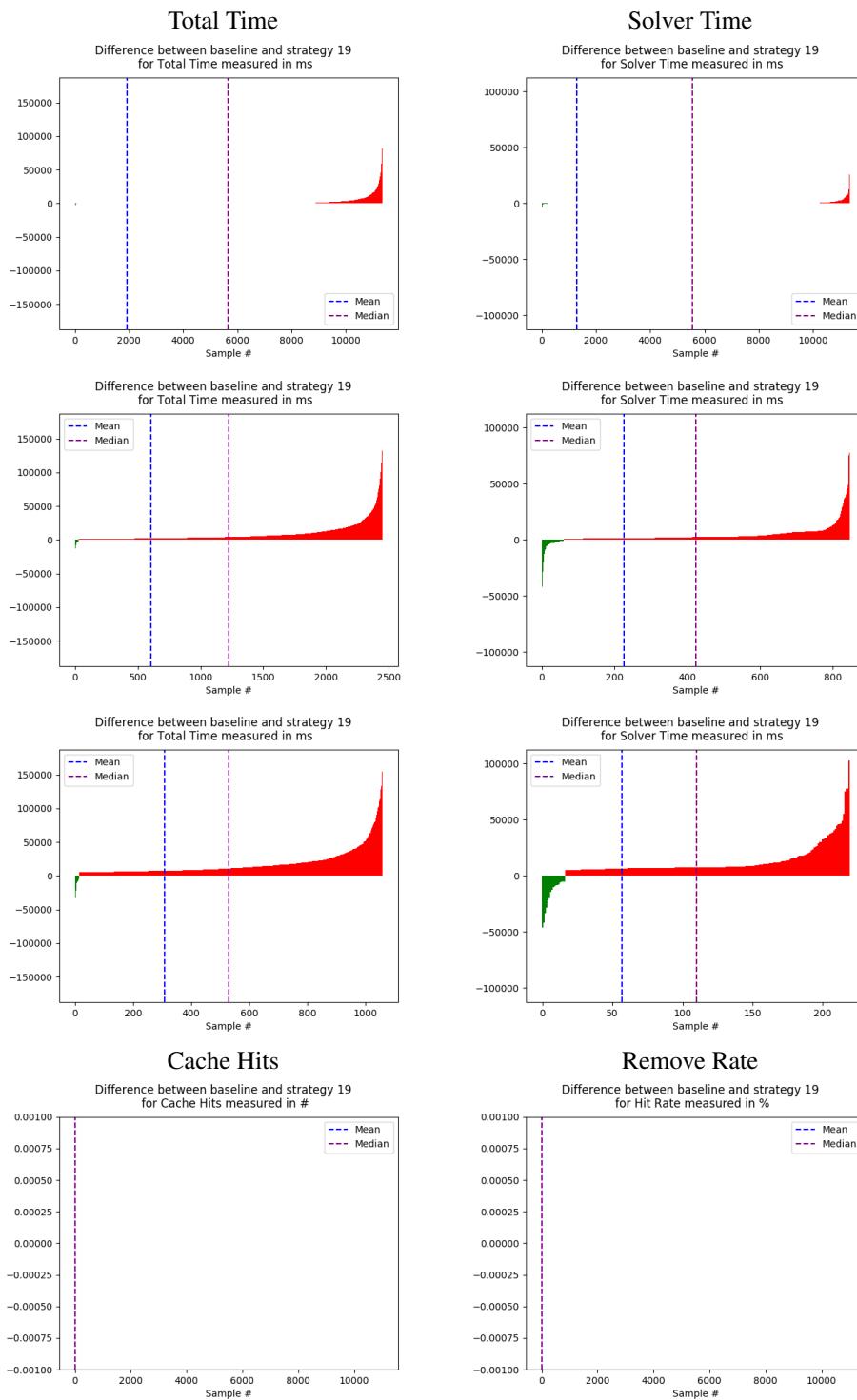
```
--cache --strat19
or
--cache --unsat-core-lookup --unsat-core-lookup-crc-full --
-lookup-when-assert
--use-rename-filter --use-shape-filter --use-range-filter
--use-forward-and-backward-filters --canonize
--variable-simplification --variable-propagation
--simplify-replaced --split-assumptions-and-assert
```

### Data Set 1 Results

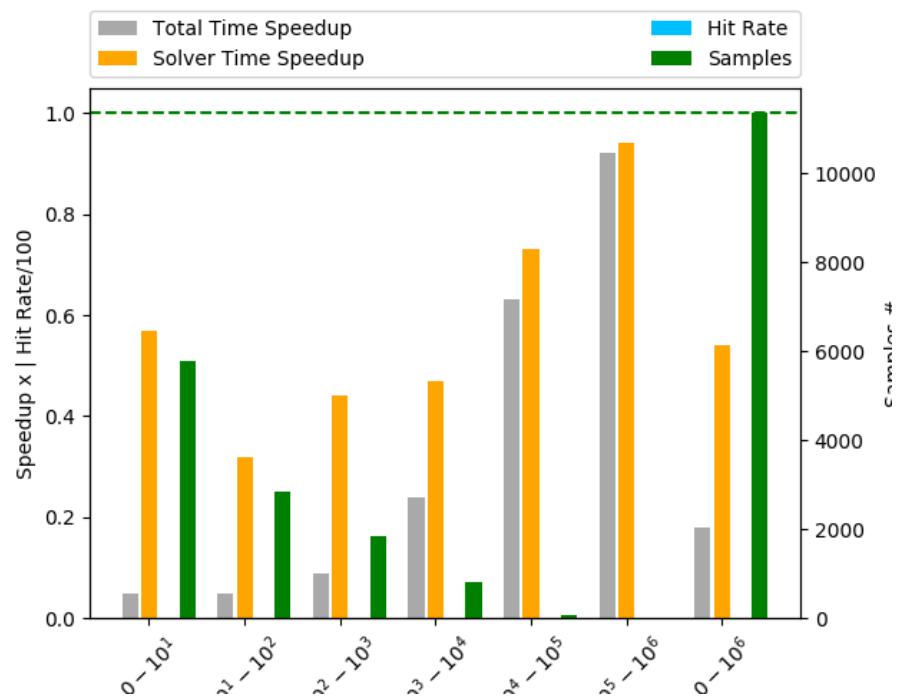
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	23307771	-46045	170566	2051.2	31.0	11363	9942	475
1000	22509228	-46045	170566	9179.95	4105.5	2452	2423	29
5000	18904521	-46045	170566	17817.64	10873.0	1061	1046	15
Solver Time (ms)								
0	4072761	-46046	102387	358.42	1.0	11363	6920	1747
1000	3593057	-46046	102387	4237.1	2436.5	848	787	61
5000	2405063	-46046	102387	10932.1	7469.5	220	203	17
Hit Rate (%)								
0	0	0	0	0.0	0.0	11363	0	0
Cache Hits (#)								
0	0	0	0	0.0	0.0	11363	0	0

## A.19. EXPERIMENT 19 RESULTS

177



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	702.63	0.05	1.78	0.57	0.00	5775
$10^1 - 10^2$	1270.41	0.05	82.05	0.32	0.00	2845
$10^2 - 10^3$	4114.89	0.09	460.32	0.44	0.00	1855
$10^3 - 10^4$	8654.03	0.24	2943.56	0.47	0.00	816
$10^4 - 10^5$	13112.10	0.63	7967.07	0.73	0.00	70
$10^5 - 10^6$	11548.50	0.92	7759.00	0.94	0.00	2
$0 - 10^6$	2051.20	0.18	358.42	0.54	0.00	11363



## A.20 Experiment 20 Results

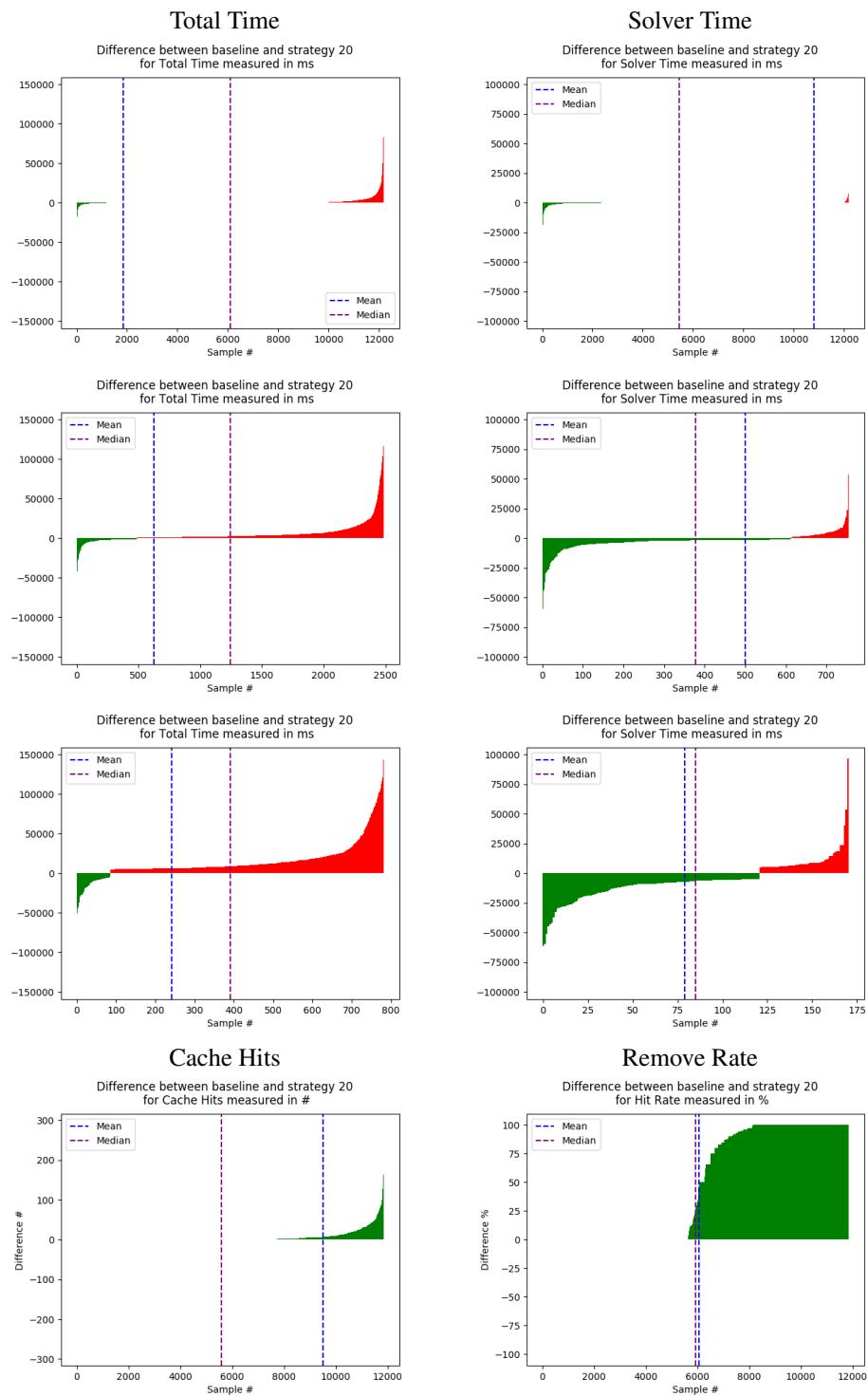
### Setup

The following commands are used for this experiment:

```
--cache --strat20
or
--cache --lookup --lookup-crc-full
--subset-lookup --subset-lookup-crc-full
--lookup-when-assert
--use-rename-filter --use-shape-filter --use-range-filter
--use-forward-and-backward-filters --canonize
--variable-simplification --variable-propagation
--simplify-replaced --split-assumptions-and-assert
--store-during-lookup --store-during-subset-lookup
```

### Data Set 1 Results

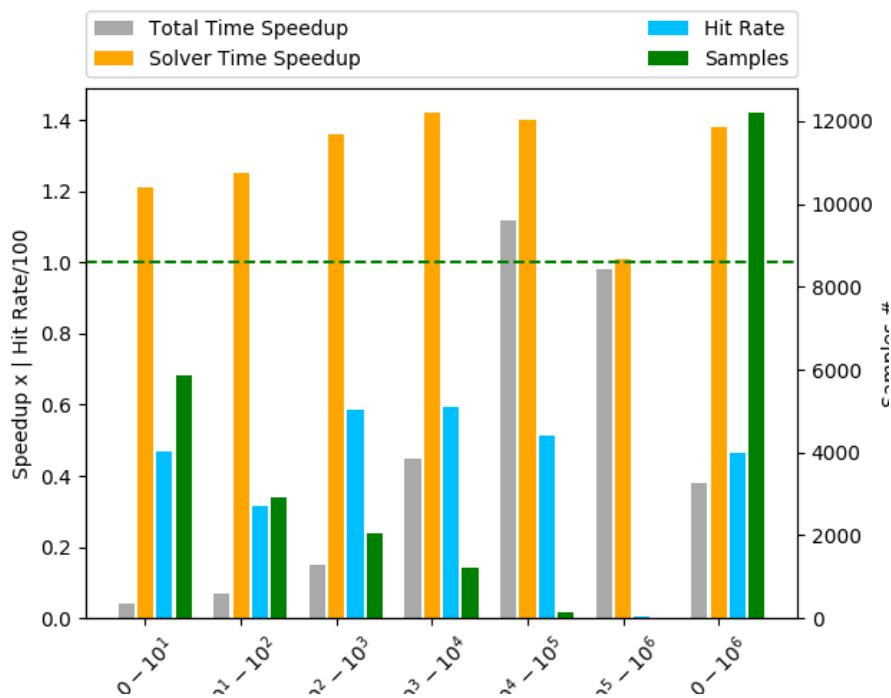
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	14465387	-61237	144790	1185.88	17.0	12198	9206	2035
1000	13791881	-61237	144790	5541.13	2596.0	2489	2001	488
5000	11388606	-61237	144790	14544.84	8925.0	783	697	86
Solver Time (ms)								
0	-2306873	-61243	96538	-189.12	-1.0	12198	1988	6756
1000	-1862264	-61243	96538	-2463.31	-1699.5	756	143	613
5000	-1015616	-61243	96538	-5939.27	-6570.0	171	50	121
Hit Rate (%)								
0	551579	0	100	46.51	26.0	12198	6227	0
Cache Hits (#)								
0	80488	0	288	6.79	1.0	12198	6263	0



## A.20. EXPERIMENT 20 RESULTS

181

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	594.78	0.04	-0.42	1.21	46.95	5861
$10^1 - 10^2$	818.79	0.07	-7.75	1.25	31.44	2904
$10^2 - 10^3$	2294.66	0.15	-93.28	1.36	58.62	2065
$10^3 - 10^4$	3495.52	0.45	-820.83	1.42	59.38	1219
$10^4 - 10^5$	-2742.86	1.12	-7391.95	1.40	51.19	147
$10^5 - 10^6$	2659.00	0.98	-1023.00	1.01	0.50	2
$0 - 10^6$	1185.88	0.38	-189.12	1.38	46.51	12198



## A.21 Experiment 21 Results

### Setup

The following commands are used for this experiment:

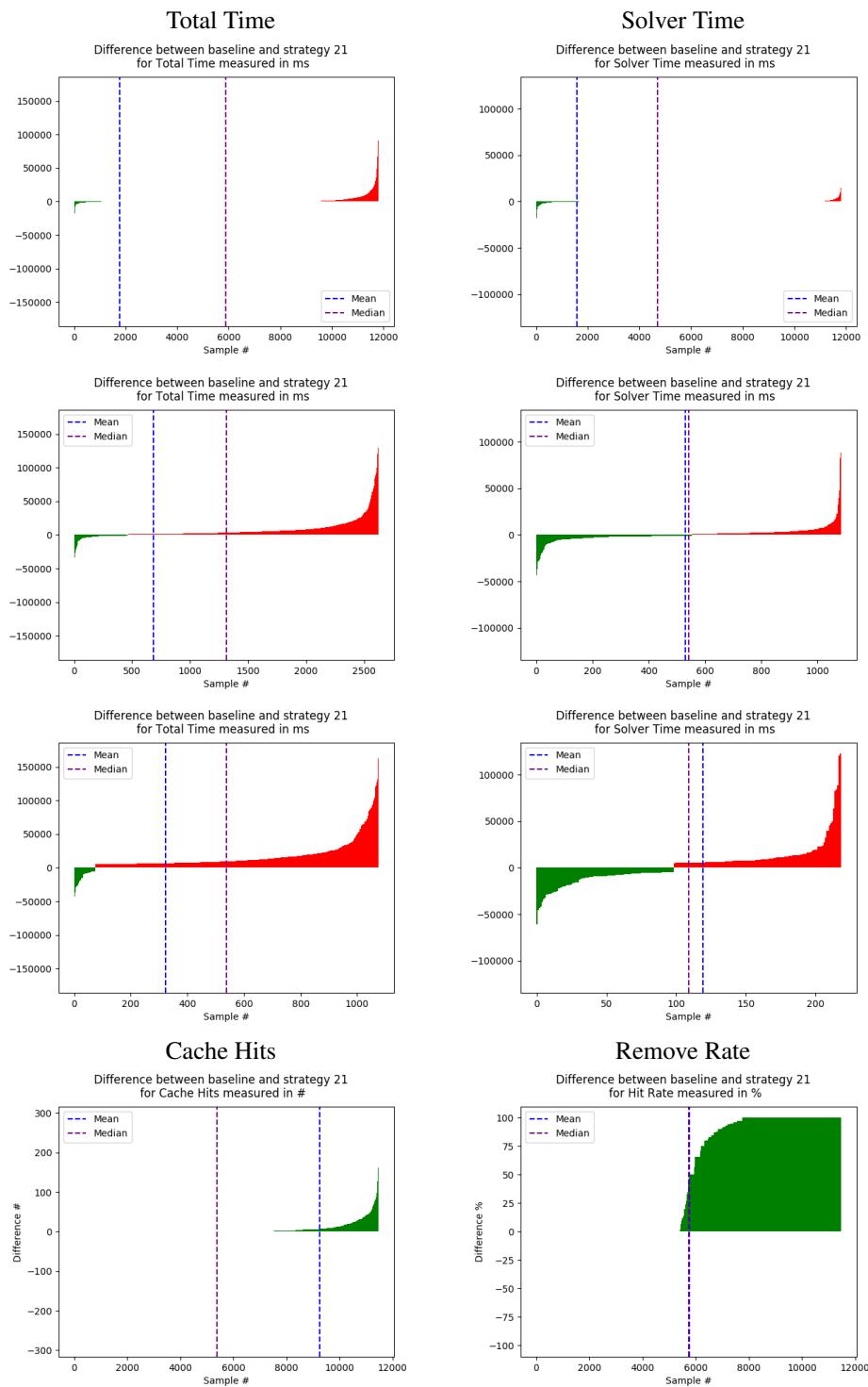
```
--cache --strat21
or
--cache --lookup --lookup-crc-full
--subset-lookup --subset-lookup-crc-full
--unsat-core-lookup --unsat-core-lookup-crc-full
--lookup-when-assert
--use-rename-filter --use-shape-filter --use-range-filter
--use-forward-and-backward-filters --canonize
--variable-simplification --variable-propagation
--simplify-replaced --split-assumptions-and-assert
--store-during-lookup --store-during-subset-lookup
```

### Data Set 1 Results

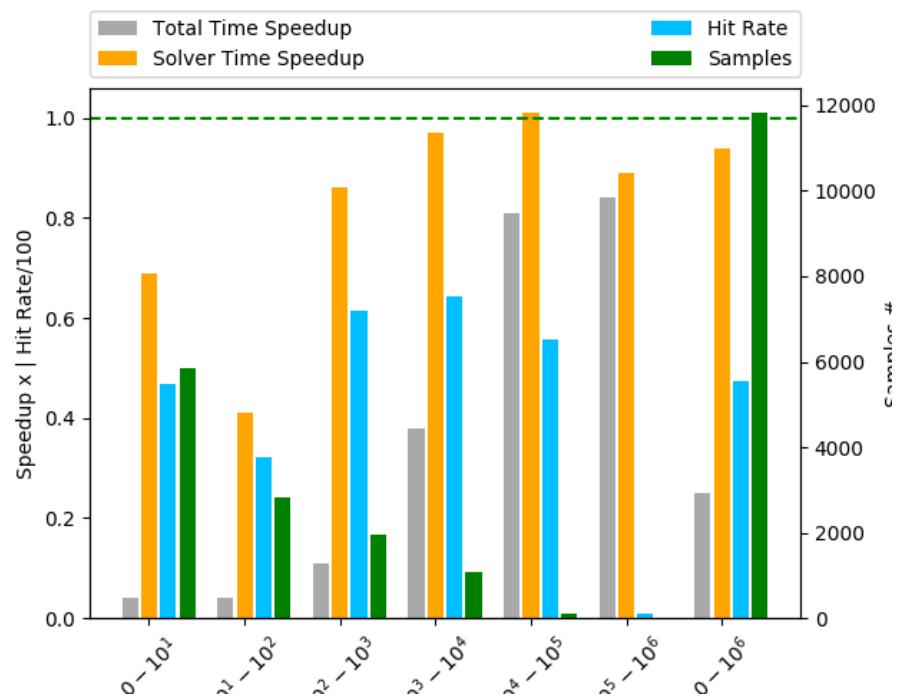
Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	19983350	-61238	169267	1690.64	24.0	11820	8818	1975
1000	19252502	-61238	169267	7325.91	3345.5	2628	2161	467
5000	17205084	-61238	169267	15975.01	9283.0	1077	1001	76
Solver Time (ms)								
0	370721	-61243	122403	31.36	0.0	11820	4695	4588
1000	470411	-61243	122403	433.56	-1071.0	1085	530	555
5000	538524	-61243	122403	2459.01	5263.0	219	120	99
Hit Rate (%)								
0	544784	0	100	47.45	45.0	11820	6063	0
Cache Hits (#)								
0	75969	0	288	6.62	1.0	11820	6099	0

## A.21. EXPERIMENT 21 RESULTS

183



TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	670.20	0.04	1.10	0.69	46.93	5850
$10^1 - 10^2$	1438.72	0.04	56.29	0.41	32.07	2830
$10^2 - 10^3$	3328.53	0.11	59.29	0.86	61.33	1953
$10^3 - 10^4$	4563.06	0.38	90.40	0.97	64.44	1085
$10^4 - 10^5$	5090.55	0.81	-256.62	1.01	55.85	101
$10^5 - 10^6$	25436.00	0.84	17062.00	0.89	1.00	1
$0 - 10^6$	1690.64	0.25	31.36	0.94	47.45	11820



## A.22 Experiment 22 Results

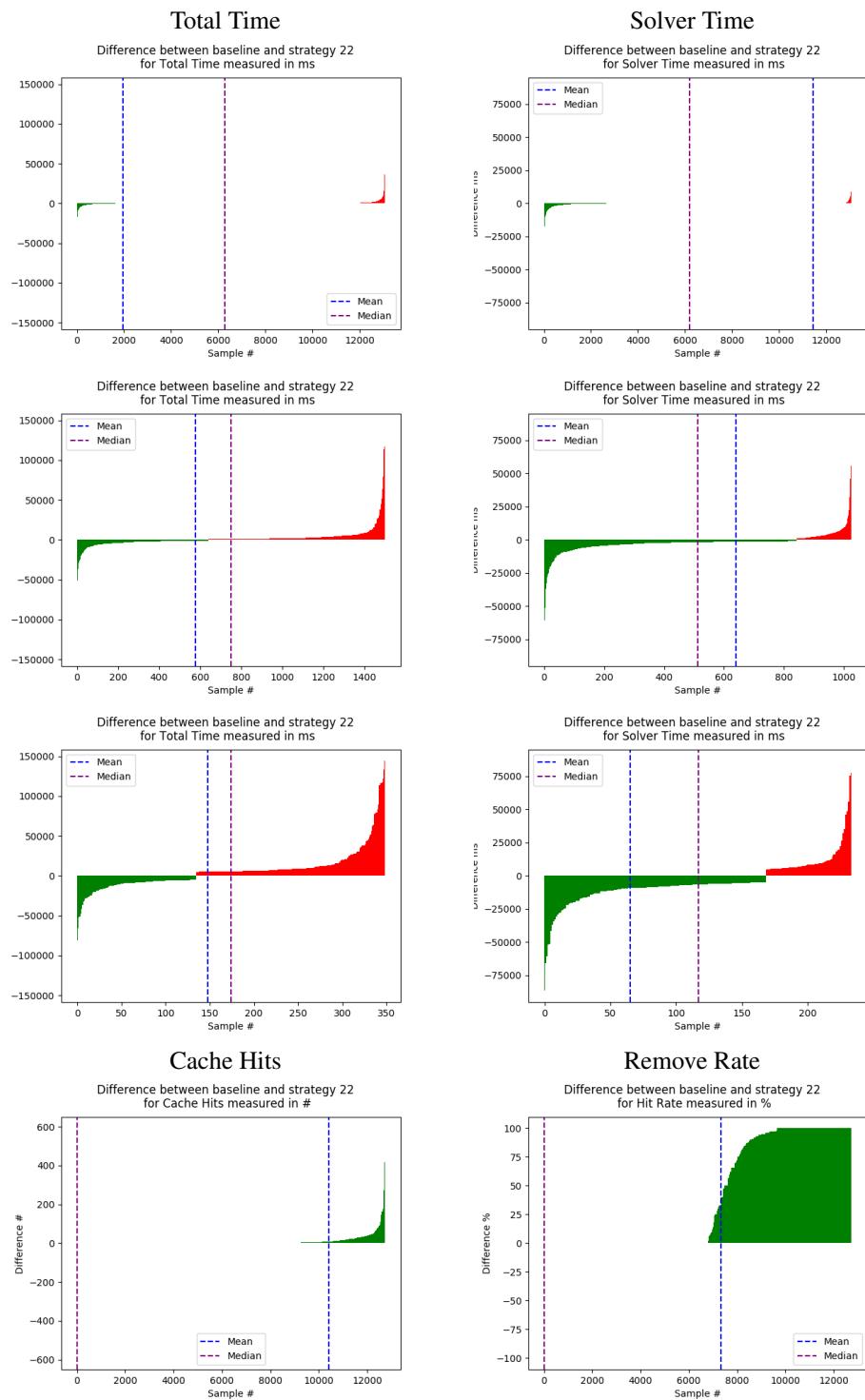
### Setup

The following commands are used for this experiment:

```
--cache --strat4
or
--cache --lookup --lookup-crc-full --variable-renaming --lookup-when-assert
--store-during-lookup
```

### Data Set 1 Results

Difference $\geq$	Total	Min	Max	Mean	Median	Samples	+	-
Total Time (ms)								
0	2716443	-81538	144242	207.65	3.0	13082	8134	3542
1000	2365990	-81538	144242	1574.18	1169.0	1503	863	640
5000	2176698	-81538	144242	6236.96	5539.0	349	214	135
Solver Time (ms)								
0	-2753803	-86614	77630	-210.5	-1.0	13082	2624	6902
1000	-2323079	-86614	77630	-2264.21	-1714.5	1026	184	842
5000	-1166224	-86614	77630	-4983.86	-6590.0	234	65	169
Hit Rate (%)								
0	507749	0	100	39.84	0.0	13082	5956	0
Cache Hits (#)								
0	121601	0	592	9.54	0.0	13082	5965	0



## A.22. EXPERIMENT 22 RESULTS

187

TR(ms)	$\Delta TT$ Avg(ms)	TT x	$\Delta ST$ Avg(ms)	ST x	HR(%)	Samples
$0 - 10^1$	409.62	0.14	-0.34	1.16	37.16	5955
$10^1 - 10^2$	134.92	0.38	-4.40	1.13	25.05	3008
$10^2 - 10^3$	292.67	0.61	-65.63	1.22	52.57	2221
$10^3 - 10^4$	-5.43	1.00	-888.97	1.40	58.81	1653
$10^4 - 10^5$	-2617.89	1.11	-4041.97	1.19	40.35	240
$10^5 - 10^6$	-28276.60	1.29	-30645.40	1.32	3.00	5
$0 - 10^6$	207.65	0.84	-210.50	1.28	39.84	13082

