Exploits in Concurrency for Boolean Satisfiability

Ali Asgar A. Sohanghpurwala

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

July 31, 2018

Blacksburg, Virginia

Keywords: Satisfiability, FPGA, SLS, MaxSAT, Parallel Local Search

Exploits in Concurrency for Boolean Satisfiability

Ali Asgar A. Sohanghpurwala

(ABSTRACT)

Boolean Satisfiability (SAT) is a problem that holds great theoretical significance along with effective formulations that benefit many real-world applications. While the general problem is NP-complete, advanced solver algorithms and heuristics allow for fast solutions to many large industrial problems. In addition to SAT, many applications rely on generalizations of Satisfiability such as MaxSAT, and Satisfiability Modulo Theories (SMT). Much of the advancement in SAT solver performance has been in the realm of improved sequential solvers with advanced conflict resolution, learning mechanisms, and sophisticated heuristics. There have been some successful demonstrations of massively parallel and hardware-accelerated solvers for SAT, but these have failed to find their way into mainstream usage. This document first presents previous work in Hardware Acceleration of Satisfiability followed by an analysis of why these attempts failed to gain widespread acceptance. It then demonstrates an alternative, hardware-centric approach, based on distributed Stochastic Local Search (SLS) that is better suited to efficient hardware implementation. Then a parallel SLS/CDCL hybrid approach is proposed that is suitable for distributed search with minimal communication overhead while maintaining completeness. Finally the efficacy and flexibility of distributed local search is considered with an adaptation to Weighted Partial MaxSAT (WPMS) and a focused case study on converted Probabilistic Inference instances.

Exploits in Concurrency for Boolean Satisfiability

Ali Asgar A. Sohanghpurwala

(GENERAL AUDIENCE ABSTRACT)

The Boolean Satisfiability (SAT) problem is an important decision problem that asks whether there exists a solution that satisfies all given constraints over a set of variables that can assume values of either 0 or 1. May real-world decision problems can be translated into SAT, and there exist efficient sequential solvers that can quickly resolve many such instances. Less progress has been made in efficiently scaling SAT solvers to modern multi-core systems and massively parallel hardware accelerators such as GPUs and Field Programmable Gate Arrays (FPGAs). This thesis explore different approaches to solving SAT based decision and optimization problems with the goal of increasing concurrency.

Dedication

To my daughters Mariyah and Husaina who were both born while I was pursuing this degree. Their incessant curiosity and drive to learn about the world around them served as a constant reminder that investigation and inquiry in the pursuit of knowledge is a defining characteristic of human nature.

Acknowledgments

Thanks to Advanced Research Computing at Virginia Tech for providing computational resources and technical support that have contributed to the results reported within this work. URL: http://www.arc.vt.edu

Thanks to my academic advisor Dr. Peter Athanas who lured me to the Configurable Computing Lab all those years ago. It has been an honor, a privilege, and perhaps most importantly, a lot of fun working with you over the years. I could always count on you for sage advice in my academic, professional, and personal life.

Thanks to Dr. Michael Hsiao, I appreciated your general advice about graduate school as well as your specific expertise and experience in Satisfiability which helped guide this research.

Thanks to Dr. Cameron Patterson who first introduced me to FPGAs and gave me invaluable advice on how to structure a dissertation centered around hardware acceleration.

Thanks to Dr. Bert Huang who wrote the Probabilistic Inference to Weighted Partial MaxSAT converter discussed in Chapter 7. Collaborating with you has been a wonderful experience, and I appreciate the time you committed to discussing, developing, and further optimizing our approach.

Thanks to Dr. Mark Jones whose invaluable advice and feedback from undergraduate coursework, my Master's thesis, and now this work has always resulted in a higher quality output product.

Thanks to my family who have been incredibly supportive and patient as this journey has stretched on. Their love and guidance made this work possible.
To my mother Nafisa who raised me with love and empathy. You will always be missed, but your legacy continues through your grandchildren.
To my wife Fatema, whose patience at times ran thin, but never ran out. It simply would not have been possible to reach this point without your support, sacrifice, and hard work in taking care of me and our children along the way. Your earnest encouragement to pursue a PhD, choose a topic, and see it through kept me on track and motivated. We've been through a lot the last eight years, but through the adversity and prosperity, there's noone else I'd rather have by my side.
To my father Ali Akbar, who perhaps foresaw me getting a PhD before anyone else did. Thank you for the support and encouragement you have provided me over the span of my life, including the final nudges to pursue and complete my Doctoral studies. Thank you for imbuing me with the work ethic, perseverance, and temperance to follow through without giving up.
To my stepmother Mariyah who offered boundless support and encouragement. I could always count on you for an optimistic view of the difficult path forward. Your assistance, advice, and hard work in helping us raise Mariyah and Husaina have been invaluable. Your efforts to organize birthday parties as well as religious and cultural celebrations allowed me to focus on my research at critical junctures.
Thanks to my in-laws, Shabbir, Fakhera, and Abdoali Shahdawala. Your help and support over the years has proven invaluable, especially in trying times. Your assistance in raising and managing our daughters as infants and toddlers took a major load off of our shoulders.

Thanks to my extended family, the grandparents, aunts, uncles, cousins, etc. who are too numerous to name but have shown us nothing but love and support.

Thanks to my friends and colleagues from the CCM Lab, you made this graduate journey so much more enjoyable and it was a great privilege to work with you.

To Dr. Tony Frangieh, I will always be grateful for your friendship and companionship. You played a large role in pulling me back for a PhD, and helped me keep things in perspective and stay motivated enough see it through.

To Dr. Jacob Couch, it's been a pleasure working with you on various projects and I value our friendship and long conversations over the years. Your long and arduous path to a PhD was certainly a motivating example that illustrated the value of perseverance.

To Dr. Kryszstof Kepa, I had a great time working with you in the CCM Lab and on the RDA tools. You were a great help during the process of developing dissertation ideas, your skeptical mind is perhaps the greatest filter of mediocre dissertation topics I've encountered yet.

To Ryan Marlow, It was a lot of fun spending time with you all these years, through the Happy Wok trips, bike rides, political conversations, and working together on the Consolidated Logic startup; it's been a pleasure.

To Shenghou Ma, I truly appreciate our long discourses on potential research topics. Somehow I always left our conversations having learned something or with a new idea or perspective. Many small implementation optimizations and thoughts about PRNGs were motivated by conversations I've had with you.

To Dr. Thaddeus Koehn, It was a pleasure working and conversing with you. I've learned a significant amount from you in terms of digital design and I wish I had learned more from you about time management.

To Dr. Jehandad Khan, it was great spending time with you and getting to know you, especially during our numerous Bollo's breaks.

To Mohammed Hassan, our early collaboration on hardware SAT solvers and the survey of existing techniques was greatly helpful to pursuing this work. I wish you the best of luck in completing your own PhD journey.

To the rest of the CCMLab folks who have helped me along the way and made my time in graduate school more enjoyable: Dr. Jorge Suris, Dr. Adolfo Recio, Charles Irick, Dr. Andrew Love, Dr. Wenwei Zha, Matt Shelburne, Prabhaav Bhardwaaj, Shaver Deyerle, David Uliana, Kurt Rooks, Chris Dobson, James Demma, Kevin Lee, Kevin Zeng, and many others. I'm still in awe of the sheer amount of talented and amazing people I've met through the CCMLab and continue to encounter in my professional career.

Thanks to the Hume Center personnel and affiliated researchers that I had the good fortune of working with. To Dr. Robert McGwier, it was a pleasure working with you and getting to learn from your experience. To Dr. Chris Headley, our interactions and collaborations were always enjoyable and I appreciated your patience working with someone not as skilled in the wireless arts. To the rest of the Hume Center staff I worked with including Alex Poetter, Dr. Zach Leffke, Dr. Joe Ziegler, Geffrey Moy, Dr. John Black, Dr. Alan Michaels, and others, thank you for your help along the way.

Thanks to the folks at the OESRC who were of great help on various projects including Jon Talerico, Tim and Melissa Leake, Chris Phelps, Nick Littier, and Warren Lucero.

Thanks to my colleagues at Graf Research. To Jonathan Graf, thanks for the opportunity to work here, the experience and exposure has been invaluable. Your flexibility, understanding, and friendship was pivotal in allowing me to complete this research while managing the complexities of a growing family. I'll try to avoid any further incidents that end with you picking me up from the hospital. The team you have assembled at Graf Research is first class, and I have much yet to learn from Dr. Scott Harper, Whitney Batchelor, Tim Dunham, Katrina States, Angela Keene, Stacey Harmond, Ed Carlisle, and Michael Capone who have all been a pleasure to work with.

Thanks to the folks I worked with at X-COM Systems and Bird Technologies Group. Thanks to my supervisor Romel Haq whose flexibility allowed me to both complete my Master's thesis and start my PhD coursework while working on tasks for X-COM Systems. Thanks to my colleagues there: Steven Renich, Nick Santana, Jason Blevins, Michael and David Erisman, Jim Taber, Steve Plante, Marty Mosier, Brandon Smithers, Troy Calderwood, Michael Fetto and of course Elena. To my friend Mike McDanield who I credit with getting me into cycling and giving me a ton of advice early on, you were taken from us too soon, and you are sorely missed.

Thanks to those I've had the privilege of working with from USC ISI/East on various projects: Matt French, Dr. Neil Steiner, Dr. Aaron Wood, Dr. Andrew Schmidt, and Dr. Travis Haroldsen, among others.

Thanks to all the friends who provided companionship through a long journey in undergraduate and graduate studies including Ashwin, Idi, Neil, Wesley, Chris, Ammar P., Adnan, Abbas, Ammar M., Daniel, Yusuf, Juzer, and Zahra among many others.

Thanks to all those in the local cycling community who I've ridden with on East Coasters, NRVBA, and even a few Wednesday Worlds rides. Thanks for letting me tag along and (usually) slow you down. To Dr. Harold Trease and Geffrey Moy who always encouraged me to challenge myself with longer and faster rides. The stress relief and modicum of fitness provided by riding with you were pivotal to maintaining my sanity during this journey.

Thanks to all the Blacksburg locals I've encountered, especially at the local restaurants, coffee shops, and businesses; you truly made this town a special place to live.

Contents

List of Figures

List of Tables

List of Abbreviations

FPGA	Field Programmable Gate Array
MaxSAT	Maximum (Boolean) Satisfiability
SAT	Boolean Satisfiability
WPMS	Weighted Partial MaxSAT

SAT is the Boolean Satisfiability Problem which asks whether there exists an assignment for a given Boolean Formula that evaluates to TRUE.

MaxSAT is a generalization of SAT that attempts to maximize the number of satisfied constraints in a Boolean Formula, even if no full solution exists.

WPMS or Weighted Partial MaxSAT is a further generalization of MaxSAT that assigns weights to each Boolean Constraint. The goal of WPMS is then to maximize the weight of satisfied constraints while ensuring that a subset of *hard* clauses are satisfied.

FPGAs are devices consisting of large arrays of reconfigurable logic resources and flexible routing architectures that allow for the arbitrary implementation of digital circuits. A major advantage of FPGAs over other architectures is a large amount of available fine-grained parallelism.

Chapter 1

Introduction

Boolean Satisfiability (SAT) is the canonical NP-complete problem[35] and, thus, holds great importance in computer science and mathematical theory. It also has a wide variety of practical applications in fields such as artificial intelligence (AI) planning, software verification, bioinformatics, and electronic design automation (EDA)[85]. Though all known algorithms exhibit worst-case exponential complexity, modern sequential software solvers leverage advanced branching heuristics, learning algorithms, and highly-optimized implementations such that many large and/or difficult problem instances are tractable and can be solved in reasonable amounts of time. NP-hard optimization generalizations of SAT such as MaxSAT and Weighted Partial MaxSAT (WPMS) also claim significant applications in Hardware Verification[31], AI Planning[133], Relational Inference[84], and Satellite Data Management [22] among others.

SAT is a well known combinatorial problem that determines whether a certain boolean logic formula representing a circuit instance can be satisfied by a truth assignment. A circuit instance is normally represented in Conjunctive Normal Form (CNF), which is a conjunction (logical AND) of a number of clauses, where each clause is a disjunction (OR operation) of literals [108]. To satisfy the whole CNF instance, all clauses must be satisfied, which means at least one literal in each clause should evaluate to "true". This problem can be solved using an obvious brute force algorithm that tries every possible combination of input assignments; however, this has exponential complexity (2^n), where n is the number of input variables. The problem can be modeled as a Binary Decision Diagram (BDD) where transformation into a Reduced Ordered BDD (ROBDD) makes it easy to determine satisfiability. The problem can also be mapped to a graph and solved using branch and bound algorithms. However, to reduce the complexity of any algorithm, heuristics and pruning techniques along with several other optimizations are used to minimize the search space.

The importance of the problem coupled with the large amount of computation involved motivated many attempts at parallelization and hardware acceleration of SAT. An excellent survey of reconfigurable hardware SAT solvers was published in 2004 by Skliarova and Ferrari[122]. Since then a multitude of SAT accelerators have been proposed with varying degrees of success; a discussion of the most prominent ones is found in[125]. Several of these published architectures made notable contributions to the state of hardware SAT acceleration; however, it is not clear that the overall advancement of hardware SAT solvers has kept pace with the significant advancements of increasingly sophisticated and competent software solvers. Algorithmic, heuristic, and implementation enhancements have led to performance gains that have scaled software performance faster than advancements in CPU performance. Software solvers are capable of solving large (millions of variables and clauses) and/or difficult problem instances in reasonable amounts of time. These advancements are easily tracked by observing the results of the annual SAT Competitions and SAT Races[4].

The open-source requirement of these competitions further enable rapid enhancements by allowing new ideas to be tested on stable solver platforms. In fact, of the five solvers that placed in the Sequential Application tracks in 2014, three of them were built off the Minisat code-base[3]. There is no such transparency when comparing hardware solver contributions, one is left to rely on the handful of benchmarks selected by the authors of the publication. With no source available for the published hardware solvers it is impossible to reproduce results for fair comparison on modern problem instances on modern hardware. Even when common development boards are targeted, binaries are not available for third parties to reproduce the results.

A successful hardware SAT accelerator thus faces the following challenges:

- Extracting significant speedup over a modern software solver is difficult; modern software solvers are capable of solving large industrial problems with millions of variables in a reasonable amount of time.
- Without a standard set of benchmarks, platforms, and availability of source or implemented binaries, it is impossible to fairly compare different hardware designs quantitatively.
- Lack of available source code for previous designs makes it impossible to build on previous work,so hardware designers always start from scratch.
- Most successful software algorithms are not well-suited to hardware implementation due to complex control structures and dynamic memory requirements.
- SAT instances generated from real problems tend to be quite large. It is essential for a practical hardware accelerator to scale gracefully in terms of area usage, memory usage, and runtime.
- SAT solver runtime is highly variable, various instance types are best solved with differing heuristics, differing algorithms, and even hybrid solvers.

With these challenges in mind, it is possible to extract a set of insights and constraints from past contributions to hardware-accelerated satisfiability to help identify what is necessary for a hardware SAT solver to make a practical contribution. A hardware solver should be able to support SAT instances that encode real problems; solving small, randomly generated or hand-crafted difficult instances have little practical value. Following from this, area usage, memory usage, and runtime should scale gracefully with the size of the problem. Approaches that scale quadratically or exponentially with problem size have limited applicability to real-world problems. If it isn't possible to scale the core solver in a linear fashion, then efficient partitioning must be used to break up the problem. Partitioning methods should allow for solving individual bins or partitions of the problem without significantly affecting runtime by high partitioning overhead. As such, architectures that partition the problem should be able to scale either partition size or support parallel evaluation of partitions to maintain speedup as instance size increases. Similarly, the runtime for each iteration of the algorithm must be bounded. If the runtime of each decision propagation scales with problem size, then hardware cannot be competitive with the predominant software algorithm where propagation time is effectively bounded. The effects of Amdahl's law should be considered when attempting to accelerate SAT. Several designs accelerated only the stable and compute-intensive Boolean Constraint Propagation (BCP) step in hardware, which is estimated to comprise 80-90% of overall execution time. This limits speedup to 5-10X. Finally, hardware architectures should be thoroughly profiled to determine what type of problem instances are well-suited to that architecture as is it is unlikely that a truly general-purpose SAT accelerator can be built. Even in the software world where Conflict Directed Clause Learning (CDCL) based solvers dominate on many industrial instances, other solver classes such as lookahead based solvers and Stochastic Local Search (SLS) solvers still perform better on different types of instances such as small hard-combinatorial problems.

1.1 Advancing the State of Concurrent and Hardware Accelerated Satisfiability

As will be discussed in Chapter 2 there have been many attempts to parallelize and accelerate Boolean Satisfiability in FPGAs as the combination of large amounts of Boolean computation, algorithms with relatively complex control structures, and opportunities for fine-grained parallelism makes this seems like a problem particularly well suited for hardware acceleration. However, none of the solutions proposed to date seem to have found widespread adoption as a SAT accelerator.

One overarching theme is that hardware implementations were frequently developed based on the most successful software solvers and algorithms. These attempts were generally not significantly faster, or they were not scalable in terms of performance or logic/memory capacity when applied to real world problems. This is largely due to the complexity of algorithms intended for general purpose processors that may not be well-suited for efficient hardware implementation. Other designs were built from the ground up to be hardware-centric but failed to consistently outperform software, required long recompilations for each problem instance, or were not scalable in terms of memory or logic requirements. Still other approaches successfully accelerated portions of SAT solving such as constraint propagation but speedups were limited by Amdahl's law to 5-6X. Indeed, recent work[75] has postulated that the structure of the resolution based proofs built by complete SAT solvers limits the amount of exposed parallelism even in the presence of an ideal information sharing framework. In accordance with Challenge 7 presented as one of the Seven Challenges in Parallel SAT Solving[61] it is likely that CDCL simply isn't well-suited for parallelization and it likely makes more sense to start from scratch with a different approach. Another challenge facing existing works is that they present themselves as general purpose SAT solvers instead of characterizing a particular class of problems well-suited to their implementation. Most SAT solvers, including the proposed HW accelerators exhibit superior performance on some instances and instance classes, and inferior performance on others. A stronger motivation for adoption could be made if hardware accelerators consistently excelled in a specific real-world application of SAT.

The purpose of the work presented here is to advance the state of parallel and hardware accelerated satisfiability through consideration of different approaches with a particular focus on an efficient and scalable hardware-centric approach that brings forth significant advantages over software in an impactful real-world application.

One approach that showed promise but had not been thoroughly explored was that of massively distributed SLS. This approach simultaneously addressed several of the challenges listed above, specifically distributed SLS threads:

- Employ simple decision metrics and control flow amenable to hardware implementation
- Do not employ clause learning techniques that would require dynamic clause database modifications
- Expose parallelism without requiring information-sharing or communication between independent threads
- Fixed logic requirements for an application-specific architecture is scalable to different memory architectures

The probSAT algorithm was chosen because it showed strong performance in SAT competitions, contained a simple control flow suitable for hardware implementation, and provided a simple C implementation well suited for porting via High-Level Synthesis. The advantage of an SLS solver is that instances run with different seeds can be run in parallel with full independence. A hardware prototype was built, featuring a many-core architecture where the same instance was run on up the 128 cores on a single Xilinx Virtex 7 FPGA. This implementation as presented in [124]showed strong performance on the small problems supported by the prototype (limited by BRAM capacity), with up to 99X speedup over the software implementation and up to 800X speedup over MiniSAT.

The above hardware prototype illustrated the potential of this many-core SLS approach, and solidified the notion already supported by the survey that local search algorithms with their simple control flows are particularly well-suited to hardware implementation. However, while this prototype showed great speedup over MiniSAT on extremely difficult hand-crafted instances, the limited capacity of the BRAM-based solver prevented testing with real-world application instances. Additionally, local search algorithms are by nature ”incomplete”, they are incapable of proving unsatisfiability; thus, making this approach by itself unsuitable for many real world applications. Applications in domains such as formal verification not only require the ability to prove instances unsatisfiable. Thus further work was justified in both evaluating the distributed local search on a system without the given memory restrictions and in designing an approach that extracts parallelism from distributed SLS threads, but still supports unsatisfiable instances.

First, in order to enable rapid prototyping of different SAT solver architectures, a scalable implementation of a parallel probSAT solver was built using a hybrid OpenMP/MPI approach and tested on the Virginia Tech Advanced Research Computing clusters. This implementation allowed for evaluating performance of this approach on larger instances, evaluate different heuristics and seeding techniques while still being able to scale to large numbers of cores. Using this implementation as a base, a hybrid solver architecture was devised that uses the local minima from each iteration of distributed local search to incrementally contribute clauses to a complete solver that can find partial solutions or prove unsatisfiability.

This approach was proposed before as an iterative single-threaded application in [50], but the work presented here is unique in allowing the local search threads to explore different parts of the search space in parallel with support for a one-way, asynchronous communication architecture that allows the CDCL solver and SLS solver threads to operate concurrently while hiding communication overhead.

Local search threads operate independently, however, it is difficult to ensure that they are not concurrently exploring the same portion of the search space. Developers often rely on platform provided pseudo-random generators to produce a random distribution of assignments across threads. This can be an effective approach and the pprobSAT solver that won first place in the SAT 2014 Competition's Parallel Random Track uses the python `random.randint()` function to seed the GNU libc `rand()` function in each of the independent solver threads. The `rand()` function is then used to set the polarity of each variable in the initial assignment. In this work more deliberate selection of seeding procedures and pseudo-random number generators are proposed that offer benefits including an even distribution of assignments across the Hamming space and in some cases better overall performance.

In observing the results from recent SAT[4] and MaxSAT[1, 2] competitions and recent publications [80] it appeared that the dominance of complete solvers was less complete in the MaxSAT and Weighted Partial MaxSAT domains. For Weighted MaxSAT in particular, local search based solvers with simple control flows and heuristics were able to find reasonable solutions for difficult real-world problems where the best complete solvers were unable to prove an optimal solution even when allocated much more time in the MaxSAT competitions (5000s vs 300s). Motivated by the strong performance shown in[80], the initial approach was to adapt the existing probabilistic solver to support weighted clauses. When this approach performed poorly, a simple greedy stochastic local search algorithm was developed with age-based tie-breakers. This algorithm performed well displayed an ability to converge towards a solution. Coincidentally, the algorithm seems to closely mirror the Ramp[49] solver that has been successful in recent MaxSAT competitions both as a standalone SLS solver and as part of the hybrid maxroster[127] solver, the winner of the incomplete track in the 2017 MaxSAT competition[2]. A scalable, distributed local search WPMS solver was built using this algorithm. The distributed solver was implemented both as a hybrid OpenMP/MPI cluster application and as an FPGA accelerator using HLS. The efficacy of this solver was proven with representative benchmarks from recent MaxSAT competitions as well as with Probabilistic Inference problems translated into the WPMS domain. The probabilistic inference problems are especially interesting as they take problems from a domain that does not seem well-suited to FPGA acceleration with typically complex algorithms and primarily floating point computations to the WPMS domain where both fine-grained and coarse-grained parallelism can be extracted and the computation is entirely in the integer domain.

1.2 Contributions

This section summarizes the contributions presented in this document.

1.2.1 Acceleration of Satisfiability with Distributed Local Search

In Chapter 3 a probabilistic SAT solver is presented that splits the search space over distributed search cores seeded deterministically. Results in Chapter 3 illustrate the efficacy of this approach in the form of a many-core solver implemented and tested on an FPGA as well as on a parallel supercomputing cluster with a hybrid OpenMP/MPI implementation.

As illustrated in Figure 1.1, this work primarily targets the class of difficult to prove satisfiable instances. This is because there is little to gain from accelerating instances that are

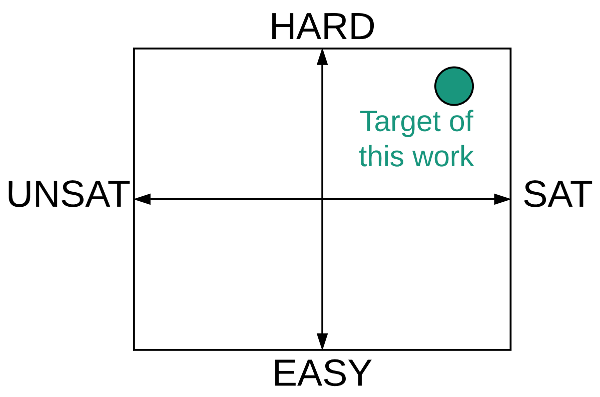


Figure 1.1: Instance Classification and Target of This Work

already easy to solve for sequential solvers, and the local search approach is unable to prove unsatisfiability. The contributions presented in Chapter 3 include massively parallel FPGA and OpenMP/MPI solvers using distributed local search. The results in Section 3.4 show how the distributed local search approach achieves significant speedups and certain classes of hard-satisfiable instances.

1.2.2 Deterministic Distribution of Search Space Across Independent Solver Cores

In Chapter 4 the argument is made for a more deliberate approach to evenly distributing the search space across local search threads. The goal of a distributed SLS SAT solver is to split the search space across threads to minimize the number of flips necessary for any search thread to find a solution. That is, the measure of a good seeding procedure the minimum Hamming distance between an initial assignment and a solution; in the general case this means the seeding procedure should result in even distribution across the Hamming space.

The main contribution presented in Chapter 4 is a simple, deterministic, and hardware-efficient scheme that maximizes the range of Hamming weights covered by the initial set of assignments while retaining guarantees about the minimum and maximum Hamming distance between initial assignments across cores. This scheme is analyzed and compared in the context of existing methods and solvers. A secondary contribution in Chapter 4 is a set of experiments that investigate the significance of statistical soundness for PRNGs used as sources of stochastic behavior in SLS algorithms. The experiments show that less complex, hardware-efficient PRNGs such as Linear Feedback Shift Registers (LFSRs) hold their own in terms of empirical performance compared to more complex PRNGS with better statistical characteristics.

1.2.3 Concurrent Completeness Without Communication Overhead

While SLS solvers can efficiently extract parallelism, one reason they are not more widely deployed in real-world applications is that they are inherently incomplete. Many real-world applications of satisfiability rely on the ability to prove unsatisfiability of an instance. On the other hand, complete algorithms that leverage Conflict Driven Clause Learning (CDCL) are able to prove unsatisfiability, but expose a limited amount of parallelism. In fact in [75] it is suggested that the depth of the resolution proofs generated by CDCL solvers presents the upper-bound value on available parallelism and speedup. This means that regardless of how efficient information sharing between parallel CDCL solvers is, the amount of parallelism is limited.

Previous work has shown performance benefits on some instance classes with hybrid SAT solvers that combine the strengths of SLS SAT solvers and CDCL solvers to create a stronger overall solver. Existing hybrid solvers either don't share data between components or share data back-and-forth in an iterative hill-climbing exercise as described in [50]. However, most of these works are focused on sequential processors where one task is running at a time. A parallel approach using OpenCL on GPUs was presented in [20], where a parallel local search was used as a heuristic for MiniSAT, but it wasn't very effective resulting in a 3X slowdown compared to MiniSAT using the standard VSIDs heuristic.

The contribution presented in 5 is an approach that combines the parallel expressiveness of distributed local search threads with a complete,incremental CDCL solver in a manner that minimizes communication overhead. The many-core SLS solver presented in Chapter 3 can be combined with any incremental SAT solver supporting the standard `ipasir` interface in an attempt to effectively climb many hills at once by searching with different seeds. At then end of one iteration (i.e. the maximum number of flips is reached), each thread communicates the unsatisfied clauses from the iteration local minima to the master thread. The master thread accumulates the set of all unsatisfied clauses, and finds a partial solution by calling an incremental SAT solver. As in [50] a random new clause is added at the end of each iteration to ensure completeness and enhance diversification. If this set is unsatisfiable, then the problem as a whole is unsatisfiable. Otherwise, the partial assignment from the CDCL solver is used to re-seed one of the SLS cores at the next iteration. This iterative process continues until either a satisfying solution is found, or the CDCL solver proves unsatisfiability. This contribution is discussed in Chapter 5 with a completeness proof and illustrations of how non-blocking asynchronous communication can be used to prevent stalls due to communication overhead.

1.2.4 Concurrent Local Search for Weighted Partial MaxSAT

Maximum Satisfiability or MaxSAT is an optimization generalization of SAT which attempts to maximize the number of satisfied clauses in the case of an unsatisfiable instance. A further generalization of MaxSAT with real-world applications is that of Weighted Partial MaxSAT [6] (WPMS). In WPMS, each clause is assigned a weight and the goal is to maximize the weight of satisfied clauses. Clauses with a weight above a certain threshold are considered *hard* clauses and must be satisfied as part of any valid solution. Clauses below that threshold are considered ”soft” clauses and are part of the optimization problem. WPMS is analogous to the more general Weighted Constraint Satisfaction problem. MaxSAT and WPMS are NP-hard[34] problems with many relevant real-world applications [22, 23, 24, 31, 70, 84, 133, 135].

As the optimization generalizations of the NP-complete SAT problem are NP-hard, they are at least as hard as the vanilla SAT problem. Indeed, a common approach for solving MaxSAT and Weighted MaxSAT problems is to decompose the instance into a series of easily solved SAT instances. The increased computation requirements for MaxSAT and Weighted MaxSAT may yield even more potential gain from acceleration and parallel processing.

In [6] a strong overview of the current state of WPMS solvers is presented. As with SAT solvers, MaxSAT solvers are either complete or incomplete. Complete solvers can be classified either as branch and bound or SAT-based. Incomplete solvers can include SLS solvers such as [81] or [80], but some of the best performers are complete solvers with the ability to print intermediate best results[6].

Investigation into distributed SLS for MaxSAT and WPMS was motivated by recent work[80] that showed how a purely probabilistic SLS MaxSAT solver can be very effective when solving harder MaxSAT instances. The similarity between the probabilistic algorithm used in [80] and the probSAT algorithm that forms the base of the many-core SAT solver presented here in Chapter 3 seemed to indicate that a similar approach could be successful in the MaxSAT domain since the only major differences were in the probability distribution function and solver parameters. While MaxSAT and WPMS have many applications in optimization and planning problems in general, many of those applications require or would benefit greatly from a complete solver. However, there are several factors that motivate the advancement of an accelerated local search solver for MaxSAT and WPMS in particular. For one there are significant classes of industrial problems including MaxCut and WCSP instances based on Earth Observation Satellite Management[22] that are not easily solvable by complete solvers, but reasonable solutions are found quickly by incomplete solvers.

The primary contribution presented in Chapter 6 is that of a distributed, scalable WPMS solver that shows exceptional performance on random and crafted instances and very strong performance on crafted maxcut instances and specific classes of industrial instances including Satellite Management[22] and relational inference.

Another area where this WPMS solver could be impactful in is in the field of Probabilistic Inference. Probabilistic inference has applications in Artificial Intelligence that allow probabilistic reasoning to be applied to real-world problems exhibiting some level of uncertainty. It has been argued that probabilistic inference may more closely mirror how animal minds solve problems [56].

1.2.5 Case Study: Probabilistic Inference

In order to motivate performance improvements, the Association for Uncertainty in Artificial Intelligence (UAI) has held competitions[55] to evaluate the effectiveness of probabilistic inference solvers. Solvers are evaluated by running them for a fixed amount of time (20 seconds, 20 minutes, or 1 hour), calculating a score, and then ranking them based on quality of results[55]. A relationship between Probabilistic Inference and MaxSAT has been shown in [105] and [11], along with the possibility that existing MaxSAT algorithms and solvers can be leveraged to more efficiently compute inference solutions. Probabilistic inference is well suited for incomplete solvers as it is often too difficult to prove optimality in the given time-window. Additionally, WPMS is a more natural domain for FPGA acceleration compared to the floating-point heavy probabilistic graphical model calculations. The contribution presented in Chapter 7 is an evaluation of the concurrent WPMS solver from Chapter 6 when applied to probabilistic inference instances.

1.2.6 Summary

Jointly, the contributions presented here illustrate a path forward for highly concurrent distributed local search based solvers for satisfiability. The presented techniques, algorithms, and implementations exploit both coarse and fine-grained parallelism, reduce learning infrastructure and inter-thread communication, and feature a less complex control flow that is efficiently implemented in a high performance computing environment including computer clusters and reconfigurable hardware. These attributes avoid the pitfalls encountered by existing solutions in terms of scalability for logic and memory capacity as well as performance on larger problems. To address some of the challenges presented in Section 1, instead of trying to compete head-on with software solvers on instances that are already easily solved, there is a focus on difficult instances and instance classes that predominant software solvers currently struggle with. Specific instance classes in the WPMS domain are identified where the presented solver excels and can meet or surpass state of the art performance either as a standalone solver or as part of a hybrid solver that uses local search to establish an upper bound.

In the process of developing these contributions, a thorough survey was performed and presented in [125] and summarized in Chapter 2, a successful distributed SLS SAT solver implementation was implemented and presented in [124] and summarized in Chapter 3. The discussion of efficient and effective distribution of the search space across cores is presented in 4, and a distributed parallel SLS/CDCL Hybrid solver is presented in 5. Finally an effective generalization of this approach to the WPMS problem in general and the probabilistic inference problem in particular is presented in 6 and 7.

Chapter 2

Background and Related Work

This chapter will first briefly provide relevant background information and summarize the current state of sequential and parallel SAT solvers with a distinct focus on recent attempts to enable massively parallel and hardware accelerated SAT solvers. For solvers targeting reconfigurable hardware in particular there will be analysis and classification of proposed architectures, identification of challenges to obtaining scalable performance benefits and adoption over modern sequential SAT solvers, and finally a path forward is suggested to address these challenges and take advantage of modern high-performance computing resources, (potentially cloud-hosted)FPGAs, and enabling technologies such as Hybrid Memory Cubes (HMCs) and High Bandwidth Memory (HBM) to enable competitive performance and scalability.

2.1 Key Concepts

Boolean Satisfiability is a decision problem that presents question of whether for a given Boolean Formula F across a set of n Boolean variables $x_1, x_2, x_3, ...x_n : \{t, f\}$ there exists a variable assignment such that $F(x_1, .., x_n) \to t$. This is traditionally represented in Conjunctive Normal Form (CNF)[108] which is a conjunction (logical AND) of clauses where each clause is a disjunction of literals, and each literal asserts the polarity of variable x_i to be positive x_i or negative $\hat{x_i}$.

2.1.1 Complete vs Incomplete Algorithms

Algorithms used to solve the Boolean satisfiability problem can be classified as either *complete* or *incomplete*. Complete algorithms are guaranteed to finish execution within a finite amount of time and decide the problem as either satisfiable or unsatisfiable [38]. They do this by methodically traversing the search space such that when the program terminates, either a satisfying assignment was found, or the entire search tree was considered and all possible branches are guaranteed to be unsatisfiable. Incomplete algorithms on the other hand are typically greedy algorithms that try to satisfy as many clauses as possible as quickly as possible, but can essentially run forever with no guarantee of finding a satisfying solution. Incomplete algorithms are also incapable of proving that an instance is unsatisfiable.

Many complete algorithms for SAT have been proposed including ones based on resolution, symbolic solving with reduced order Binary Decision Diagrams (BDDs), inference, and exhaustive search[39]. However, most successful modern SAT solvers use a combination of backtracking search and inference[39].

The primary complete algorithm for search in SAT is the Davis Putnam Loveland Longemann (DPLL)[43] algorithm. This backtracking search algorithm keeps track of assignment decisions in a binary decision tree. Variables are chosen and assigned a value of (0 or 1) in order to incrementally build the tree. After a variable is assigned, the next step is *Boolean Constraint Propagation* (BCP). This process propagates the newly assigned decision throughout the undefined clauses and determines if there are any new *implications*, *conflicts*, or *satisfied clauses*. If a conflict is detected the solver backtracks to the last decision step and flips the last assignment to pursue the opposite decision branch. Eventually the search finds an assignment that satisfies all clauses, or it has backtracked to the root assignment after exhausting all possible branches, thus proving unsatisfiability. The majority of successful solvers today implement an inference-based enhancement to DPLL known as Conflict-Driven Clause Learning (CDCL) that adds the ability to learn new clauses and non-chronologically backjump within the search. Modern solvers augment this algorithm with extremely efficient BCP built on lazy data structures, low-overhead decision heuristics such as VSIDS, and conflict resolution with non-chronological backtracking to rapidly prune the search space[86]. These advancements allow modern software solvers to process problems with millions of variables and unimaginably large search spaces in a matter of minutes or hours.

There are a number of incomplete algorithms, but most today are based on Stochastic Local Search (SLS). Local search algorithms use *Metaheuristics*, higher-level procedures that efficiently traverse the *search space* of a problem using a set of strategies that limit how much of that search space is visited[27, 33]. The strategies or heuristics used to make decisions often attempt to create a balance between *intensification* and *diversification*[27]. A greedy heuristic will intensify a solution until the search finds a local minima with no possible move that improves the cost function. On the other hand, a strategy that makes purely random decisions may do a good job of diversifying the search and visiting distant parts of the search space, but may or may not improve solution quality. Local search methods make iterative changes or *moves* within a *neighborhood* of possible moves in an attempt to find a solution to the problem[33].

SLS Solvers for SAT typically combine a greedy decision heuristic with some sort probabilistic strategy to introduce diversity. The neighborhood in this case is the set of assignments that can be reached by flipping a single variable assignment and the decision procedure selects the flip candidate. The heuristics used typically include some sort of cost function that attempts to maximize the number of newly satisfied clauses or *makes*, minimize the number of newly unsatisfied clauses or *breaks*, or maximize the value of $makes - breaks$. One of the earliest successful local search algorithms for SAT was the Greedy SAT (GSAT)[119] which performed a purely greedy local search. This was followed up by the WalkSAT[120] algorithm that also incorporates a random walk where at each decision point a random decision is made with probability p. It was later shown in [66] that local search algorithms incorporating random walks in this manner are *probabilistically asymptotically complete*; given an infinite runtime it is almost certain that a solution will be found by such an algorithm if it exists. More recently a new class of SLS solvers were introduced[14, 15] that make decisions based directly on probability distribution functions dependent on make or break values. These solvers such as probSAT[15] are very effective, especially on random instances winning Gold medals in the Sequential random track of the 2013 SAT Competition and the parallel random track in the 2014 SAT Competition[4]. SLS solvers in general can be simply and efficiently implemented, and are capable of finding satisfiable solutions through rapid exploration of the search space. They are however, unable to prove a formula is unsatisfiable and are clearly outperformed by CDCL solvers on many classes of real-world instances. The performance of SLS algorithms on various instance classes is highly dependent upon tuning and can be very effectively tuned automatically as shown in [76]. SLS algorithms can also be directly applied to the MaxSAT problem where the goal is to maximize the number of satisfied clauses in an unsatisfiable instance.

There have been attempts to create *hybrid* solvers that combine the strengths of CDCL and SLS solvers[50]. More recently, as seen in the SAT Competition 2014 [21] there is a tendency to use hybrid solvers that run incomplete SLS algorithms for a fixed amount of time to quickly find a solution, and then switch over to a complete CDCL-based solver if no solution was found. This allows one to leverage the strengths of both types of solvers, and such a solver won the Hard Combinatorial SAT Track in 2014[3].

MaxSAT and Optimization Generalizations of SAT

SAT can be generalized into optimization problems such as Maximum Satisfiability or *MaxSAT*. The MaxSAT problem attempts to maximimize the number of satisfied clauses within a SAT instance. Further generalizations of MaxSAT include *Partial MaxSAT* and *Weighted Partial MaxSAT*(WPMS). In these generalizations, some subset of the clauses are considered “hard” and some are considered “soft”. Hard clauses must be satisfied as part of any valid solution, but the goal of the optimization problem is to maxmimize the number of satisfied soft clauses. The WPMS variant further adds weights to each clause where the goal is to satisfy all hard clauses, and maximimize the weight across satisfied soft clauses. A good overview of traditional sequential MaxSAT algorithms with a focus on complete branch and bound techniques is presented in [78], and modern sequential and parallel solvers based on linear search or core-guided techniques are discussed in [33].

Instance Specific vs. Application Specific

Software SAT solvers are typically tuned to perform well on specific instance classes or for specific tracks in the yearly SAT competitions. However, they are still considered general-purpose SAT solvers that can be applied to any SAT instance.

SAT solvers targeting hardware accelerators on the other hand are typically limited in scope due to the fixed processing structures and limited hardware resources. Early approaches to hardware acceleration leveraged the fact that a boolean formula describes a circuit and would synthesize boolean logic formulas directly into an optimized hardware circuit in an *Instance Specific* approach[122]. These approaches such as [137] often boasted impressive run-time performance improvements as they were able to extract large amounts of parallelism by reducing the BCP step to a single clock cycle. However, the lengthy synthesis and implementation time typically negated any potential speedup.

In order to create practical hardware accelerators that did not simply push the problem complexity and execution time onto the EDA tools[129], more recent attempts at hardware accelerators take an *Application Specific* approach where as long as a problem instance fits within certain implementation bounds, they can be loaded onto the hardware by programming registers or memory whether on-chip or off-chip. Even as of 2004 [122] most research shifted to development of application specific solvers to avoid the computational expense of instance-specific placement and routing. All the approaches published since then have been application specific, and mapped instance circuits into registers, block RAMs (BRAMs), or off-chip DRAM instead of trying to encode it into the FPGA.

Heterogeneous vs. Full Hardware Solutions

The approaches discussed in [122] were loosely-coupled systems where some preprocessing was done in software but actual solver operation was fully implemented in hardware. Several solutions presented since then [41, 42, 128, 129, 134] have been heterogeneous solutions that depend on software operation for control and conflict resolution, whereas others solve the entire problem on the FPGA after some software preprocessing operations [57, 58, 71, 72, 73, 111].

2.2 Evolution of Software Solvers

2.2.1 Sequential SAT Solvers

Since the seminal 1962 paper [43], which introduced the Davis-Putenam-Longemann-Loveland (DPLL) algorithm, DPLL has been the prominent algorithm used for complete SAT solvers.

While there have been many significant enhancements since then, most successful complete solvers use the basic concepts of DPLL at their core.

In [87], the concepts of Conflict Driven Clause Learning (CDCL) and non-chronological backjumping were introduced. The search follows the same pattern as DPLL; however, when a conflict occurs, "look-back" techniques are used to determine the core-cause of the conflict. With this analysis, a new constraint can be added to the problem that prevents re-traversing unsatisfiable subtrees and the search can jump non-chronologically back to the root cause of the conflict, avoiding the sequential backtracking performed by DPLL.

The CDCL concept was further refined in[96] where the concepts of two-watched literals and the VSIDs heuristic were introduced with the Chaff solver. Both of these are key concepts which contribute to the impressive performance of modern SAT solvers. Two-watched literals is the process of "watching" two literals in each clause and only processing a clause if one of the watched literals in those clauses changed. This has a major benefit in significantly reducing memory accesses, which is a major contribution to the ability of modern solvers to solve large problems as it helps optimize caching behaviour[96]. Hardware solvers have largely avoided implementing this scheme due to its complexity. Most rely instead on storing clauses in high-speed on-chip BRAM, limiting the size of the instances that can be solved. The second major contribution of the Chaff solver is the VSIDs heuristic which is generally considered to be one of the stronger general-purpose variable decision heuristics. The basic idea is that the priority of a variable is a decaying value based upon the number of occurrences of the variable within the formula. Variable priority gets incremented if a conflict clause is added that contains that variable and decays according to a pre-set time constant. This powerful heuristic has been essential in pruning the search space. The dynamic reaction to variables that are frequently and recently contributing to conflicts allows quicker resolution of local conflict scenarios versus the traditional static ordering. Dynamic ordering of variables is also something that is more complex to implement in hardware and hasn't been seen in full-hardware accelerator designs.

Sequential SAT solvers were further advanced by solvers such as MiniSAT that offered a clean, extensible, and yet extremely performant implementation of the previously discussed enhancements as well as contributions such as an effective preprocessor[45] and the use of hardware synthesis optimization techniques such as Directed Acyclic Graph optimization (DAG) to minimize the circuit before solving[47]. MiniSAT also introduced the concept of periodically deleting learned clauses based on conflict activity which proves effective in maintaining propagation performance while keeping the most pertinent learned clauses. MiniSAT's performance has been eclipsed in recent years, but its clean, performance-optimized implementation and modular code-base make it the de facto base for new solvers. In fact, in the 2014 SAT Competition, six of the nine places in the Sequential Application tracks were claimed by solvers based on the MiniSAT core[3]. Minor heuristic tweaks such as restart frequency and differences in learned clause deletion management have major performance implications that differ even based on whether an instance is SAT or UNSAT[100].

2.2.2 Parallel SAT Solvers

The most popular and effective approaches to Parallel SAT solvers leverage the sophisticated sequential CDCL techniques in a *task-parallel* manner. Task decomposition can be done either via *search-space partitioning* or with a *parallel portfolio* approach. In search-space partitioning, different branches of a DPLL search tree are allocated to separate processors. The parallel portfolio approach executes different search strategies and heuristics on different processors. In practice, portfolio solvers have been more popular and effective; they are often able to exhibit super-linear speedup when one of the cores quickly stumbles upon the correct solution. ManySAT[62] (based on MiniSAT) popularized the parallel portfolio approach with strong performances in the 2008 SAT Race and 2009 SAT Competition[65]. This solver concept was further refined by Plingeling[25] which follows the same concept as ManySAT but reduces the clauses shared between threads to unit lemmas. Plingeling has won Silver or Gold in the Parallel track for all SAT Competitions since 2011[4]. A more successful spin on search-space partitioning is the Cube and Conquer Approach[63] that uses a lookahead solver with a *cutoff heuristic* to identify branches of the search tree that can be quickly solved by fully independent parallel CDCL solvers. This approach has been implemented to great effect in the Treengeling solver which won the 2016 SAT Competition Parallel track and had second and third place finished in 2013 and 2014 respectively[4].

Several works attempted to accelerate satisfiability with parallel local search. In [9] the authors evaluated different methods of sharing information between local search threads to avoid redundant work with near-linear speedup up to four cores and good speedup up to eight cores. In [7] the same group evaluated the scalability of this approach on a compute cluster and found performance improved typically up to 16 cores and then tapered off. In [8] scalability of parallel local search algorithms with no information sharing was investigated. They found that scalability was near-linear up to 512 cores for instances in the crafted and verification families and that often the best sequential algorithm was not the best performer on a massive scale. Other instances from the random and quasigroup instance classes did not show scalable performance.

Broader overviews of parallel SAT solvers can be found in [65], [89], and [18]. A survey of parallel local search methods is found in [33].

2.2.3 GPU SAT Solvers

There have been many attempts to accelerate SAT using Graphics Processing Unit acceleration with limited success. As noted above, parallel software solvers leverage sequential algorithms with complex control and branching decisions in a *task-parallel* manner with search-space partitioning or portfolio approaches. On the other hand GPUs are designed for and excel at *data-parallel* computation where simple operations with minimal branching decisions are performed on elements of a large data set in parallel; the architecture of GPUs makes existing parallel SAT solver techniques inapplicable[36]. GPUs may be better suited to different, highly parallel algorithms with simpler control structures but the tradeoff in heuristic complexity and learning capability can leave them slower than contemporary sequential software[94]. Some works show speedups compared to sequential versions of in-house algorithms[36, 52], but lack direct comparisons to accepted benchmarks like MiniSAT. It is unclear whether these solvers would be competitive with contemporary sequential software solvers.

In [20] a hybrid solver was presented that implemented a simple parallel local search algorithm on an Nvidia Tesla GPU using OpenCL and used the local search result as a heuristic for MiniSAT. While they were able to extract significant parallelism in their local search with an order of magnitude speedup over their sequential algorithm, overall performance was still 3.6X slower than the unmodified sequential MiniSAT with VSIDs heuristic.

2.2.4 Advancements in Sequential and Parallel Optimization Generalizations

Optimization generalizations of satisfiability such as MaxSAT and Weighted Partial MaxSAT are NP-hard problems[34], potentially offering even more opportunity for exploitation of concurrency.

An overview of parallel complete algorithms for MaxSAT and its generalizations including partial and weighted partial MaxSAT is presented in in [83] along with a summary of the work originally presented in [92]. Complete MaxSAT solvers commonly use iterative calls to SAT solvers to exhaustively search the possible weight assignments[83]. *Linear Search* algorithms insert relaxation variables into soft clauses that will be set true if the original soft clause is not satisfied as part of a SAT solution. The cost of a solution can then be calculated by summing the relaxation variable assignments; the initial solution weight where all soft clauses have relaxation variables defines an upper bound to the solution. The linear search solver then defines a new pseudo-Boolean constraint on the upper bound and encodes it into the SAT instance; it stipulates that any new solution must cost less than the current upper bound. Thus, if a solution is found for the working formula, the solution cost will be less than the current upper bound. If the working formula is proven UNSAT, then the current upper bound value is the optimal solution. Lower-bound search algorithms also exist that start with a cost constraint of zero and incrementally solve SAT instances until the working formula is satisfiable. *Unsatisfiability-Based* algorithms leverage the underlying SAT solver's ability to identify unsatisfiable cores of clauses to more selectively insert relaxation variables more efficiently manage lower-bound constraints[83]. The parallel MaxSAT solvers discussed in [83] use either or both of the above strategies and split the available workers across some combination of upper bound and lower-bound search and exchange information regarding bound values and learned clauses to minimize redundant work. Similar to parallel SAT solvers, parallel MaxSAT solvers can be classified as portfolio solvers or search-space splitting solvers. Portfolio solvers use different algorithms, or different configurations of the same algorithm to work in parallel on the same instance and share data to reduce redundant work. Search-space splitting based solvers allocate workers to search different bound intervals or different portions of the search tree. Since these solvers use underlying CDCL SAT solvers, they can also benefit from sharing learned clauses with each other, though doing this safely and scalably turns out to be quite complex. One issue that comes up due to the vast search space of MaxSAT, is that even the space of provably optimal solutions in terms of weight is quite varied and a non-deterministic solution is unsuitable for applications where reproducability is important[83]. In [91, 92] a deterministic parallel solver approach was proposed that requires structured synchronization between the search threads to ensure consistency between threads in terms of lower-bound values. They found that despite the synchronization overhead, performance was comparable between deterministic and non-deterministic algorithms on difficult problem instances. Ultimately while these approaches to show performance increases over sequential MaxSAT solvers, they are not very scalable with the maximum reported speedup in [92] being 2.57X with eight threads. In [97] a distributed solver is presented that uses either Guiding Paths or Search Space Splitting approaches in a cluster environment and shows that increasing the number of threads up to 16 does increase the number of solved instances. They note that gains are small because solvers quickly converge to near-optimum values quickly which limits the diversification possible within the search space.

There have been many recent works on local search for MaxSAT and WPMS[49, 80, 81, 82]. In those works as well as in recent MaxSAT competitions[1, 2] local search solvers have shown strong performance, especially on crafted and random instances. For Weighted MaxSAT in particular, local search based solvers with simple control flows and heuristics were able to find optimal or near optimal solutions for difficult industrial instances including spot5[22] and rna alignment[88] instances as well as crafted problems such as MaxCUT instances where the best complete solvers were unable to prove an optimal solution despite being allocated 60X as much time(5000s vs 300s) in recent MaxSAT competitions[1, 2]. Hybrid solvers incorporating local search and complete MaxSAT algorithms also show very strong performance where a local search solver is typically run for a fixed timeout to generate a reasonable upper bound value before starting the linear search process. The MaxRoster[127] solver which uses Ramp[49] to generate the initial upper bound dominated the incomplete track in the 2017 MaxSAT competition[2] despite being a complete solver.

Other hybrid approaches to MaxSAT have performed well in recent MaxSAT competitions. These include [114] and [40] that combine Mixed Integer Programming optimization with iterative calls to SAT solvers. The MaxHS solver based on [40] won the Weighted MaxSAT category in the 2017 competition[2]. An approach that combines Decimation with local search was presented in [30] and exhibits state of the art performance on all classes of unweighted MaxSAT instances as well as Random and Crafted WPMS instances. Due to the iterative nature of these algorithms, it is unclear that this could be effectively parallelized.

A hybrid solver with potential for parallel operation was proposed in [77] that used MiniSAT to guide a local search procedure by restricting the search neighborhood to variables whose value has not been fixed (decided or implied) by the MiniSAT search. This approach was shown to be effective in finding MaxSAT solutions to hard UNSAT instances from earlier SAT competitions. However, this approach is not effective on real-world MaxSAT instances as MiniSAT quickly proves those to be unsatisfiable and falls back onto pure local search. The authors of [77] postulate that this approach would be well-suited to multi-core architectures, it isn’t clear that they actually tested speedup with multiple threads.

2.3 Hardware Accelerated Satisfiability

2.3.1 Legacy Hardware Acceleration Approaches

It is beneficial to briefly summarize the conclusions from [122] so as to be able to contrast it with advancements since then. Skliarova and Ferrari covered 26 papers discussing hardware SAT accelerators, encompassing half that number of unique solutions. They summarized the analyzed approaches in terms of *Algorithmic issues*, *Programming Model*, *Execution Model*, *Reconfiguration Mode*, and *Logic Capacity.* The algorithmic issue discussion encompassed whether a solution was DPLL, PODEM, or SLS based. Programming Model addressed whether a solver was instance or application specific. The execution model analysis focused on hardware/software partitioning of the solvers and whether hardware/software partitioned solvers divided work according to computational complexity or logic capacity. Reconfiguration Mode addressed whether a solver used a single fixed-function bitstream or reconfigured the device (partial or full reconfiguration) to modify FPGA operation at runtime. Finally, Logic Capacity dealt with the various approaches of solving instances that required too much logic for available hardware.

The approaches presented in [122] cover the gamut of the above classifications. There were instance-specific approaches that synthesized the entire CNF as a circuit and used an array of state machines to perform basic DPLL procedures[106, 136] and there were application-specific designs that leveraged dynamic reconfiguration for context switching on larger problems[44]. The most notable design according to Skliarova and Ferrari was [137] where an instance specific implementation was used that allocate clauses among a pipelined network of processing elements that can each process a fixed number of clauses with a control unit at the end of the pipeline that manages the SAT solver operation. This work also presented support for non-chronological backtracking as well as conflict resolution and clause learning. This design was quite advanced for its time, and some concepts were emulated by other later solvers. The design presented in [44] also earned a special mention for its use of partial reconfiguration to context switch between hardware pages, allowing for large formulae to be processed.

It was noted that early SAT solvers were almost overwhelmingly instance specific but any potential speedup was likely negated by implementation time on most non-trivial problems. As a result, the authors suggest that future solvers will be predominantly application specific as were most contemporary hardware solvers.

Skliarova and Ferrari exhibited clairvoyance in noticing some trends that persist in the discussion here today. One is that it is difficult to analyze and compare the performance of the various hardware solvers given the lack of standardized benchmarks and comparison platforms when comparing to software.There is also a lack of transparency in the benchmarks, and some authors ignored or failed to present hardware compilation and reconfiguration time when considering performance. They also noted that while some works presented impressive speedups of up to several orders of magnitude, this speedup calculation is beholden to benchmark selection and often speedup is shown on instances with little practical importance. Also, most designs compared themselves to GRASP, which at that time was already obsolete; far more advanced software solvers such as Chaff and zChaff were already available. Finally, they note the then-recent advancement of hardware-software partitioning to allow for the solving of much larger problems.

2.3.2 Modern Hardware Acceleration Approaches

Prior to explaining the key concepts and details regarding the various solvers, a brief overview of the higher-level classifications is presented here. The main classifications presented in this

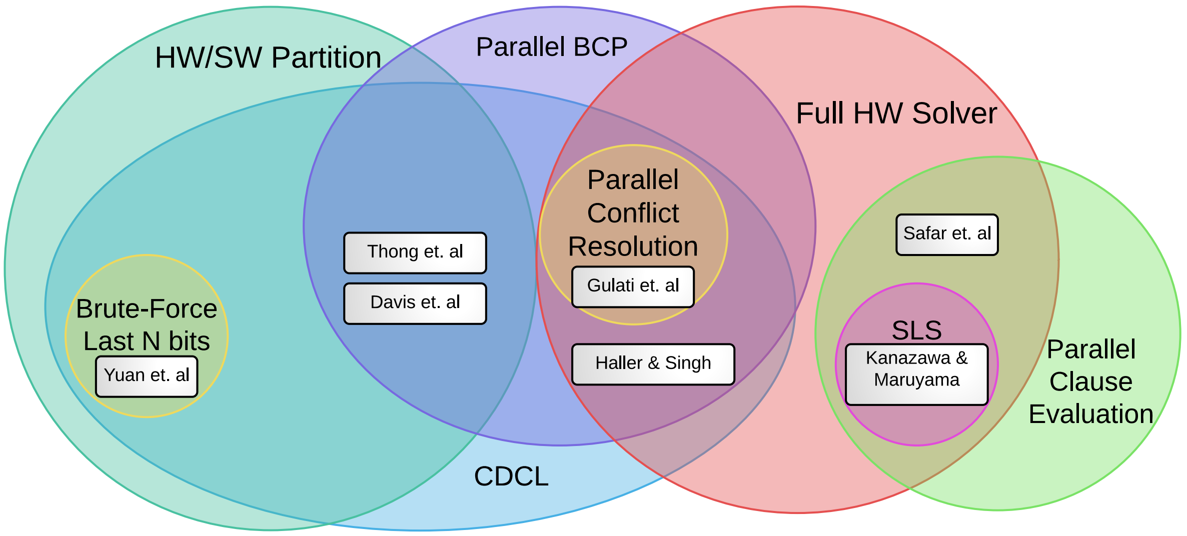


Figure 2.1: High-level classification of Modern SAT Accelerators

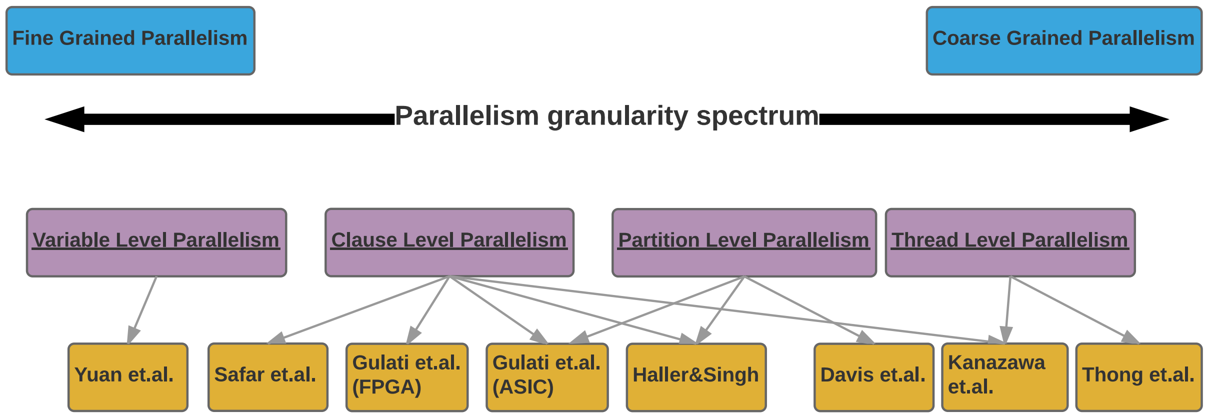


Figure 2.2: Granularity of Parallelism in FPGA SAT Solvers

section are granularity of hardware parallelism and HW/SW partitioning. The majority of solvers presented here implemented the CDCL algorithm either fully in hardware or as a BCP co-processor assisting a software solver. Safar et al [111, 113] opted to create a custom strategy based on the seminal Davis-Putnam-Logemann-Loveland (DPLL) algorithm with a less control-heavy conflict resolution strategy that is efficiently implemented in hardware. Kanazawa and Maruyama build a SLS solver based on the WalkSAT algorithm. An overview of these classifications is shown in Figure 2.1. Most FPGA accelerators aim to extract speedups by exploiting high-levels of parallelism; the works surveyed here extracted parallelism at various levels of granularity as shown in Figure 2.2. The following levels of parallelism are defined in order of granularity as below:

- *Variable Level Parallelism* - Testing multiple variables in parallel
- *Clause Level Parallelism* - Evaluating, or propagating multiple clauses in parallel
- *Partition Level Parallelism* - Partitioning SAT instances into sub-instances that can be evaluated in parallel
- *Thread Level Parallelism* - Having multiple threads in parallel for a single instance

Heterogeneous Solvers

Since the process of deciding variables, analyzing conflicts, and learning clauses is often control heavy and complex, some solvers opt for a heterogeneous approach where a hardware co-processor accelerates computationally intense tasks for a software solver. BCP is a popular candidate for acceleration as it is the most computationally intense portion of the DPLL algorithm and is estimated to consume 80-90% of solver runtime[129]. BCP is well suited to FPGA acceleration due to the lack of data dependencies between clauses as well as memory access requirements that are well suited to BRAM. Acceleration of BCP is practical and efficient, but overall speedup is limited to around 5-10X according to Amdahl's law. The major benefit of accelerating only the BCP process in hardware is that instead of competing with advanced software solvers, FPGA-based solutions can supplement them to take full advantage of algorithmic advancements.

Table 2.1: Classification of Modern HW SAT Solvers

Solver	Completeness	Algorithm	HW/SW Partition
Safar et al [111, 112]	Complete	DPLL based	Full HW
Gulati et al [57, 58]	Complete	DPLL/CDCL (GRASP)	Full HW
Haller, Singh [59]	Complete	CDCL/Two watched lit.	Full HW
Davis et al [41, 42]	Complete	CDCL	SW Solver, HW BCP Coprocessor
Yuan et al [134]	Complete	CDCL/Brute force	SW Solver, HW brute-force last 13-16 bits
Thong et al [128, 129]	Complete	CDCL	SW Solver, HW BCP Coprocessor
Kanazawa et al[71, 72, 73]	Incomplete	SLS (WalkSAT)	Full HW

Full Hardware Solutions

Some solvers implement the entire solver algorithm in hardware. This means that all decision, control, and heuristic, and propagation logic is implemented in hardware. Some approaches attempt to map successful software algorithms into parallel accelerators [57, 58, 59] and others devised hardware-centric designs that trade algorithmic complexity for hardware efficiency[111, 113].

2.3.3 Analysis

In this section, high-level analysis is presented on the surveyed work including generalized insights, and a discussion of how some of these approaches map to state of the art hardware and SAT instances. Table 2.1 shows the basic characteristics of solvers surveyed in this chapter, which presents the classification criteria most fitting for modern solvers.

Old & New Classification

Interestingly, the various reconfigurable SAT solvers published since the 2004 survey [122] seem to have converged to derive common answers to certain design issues in a way that obsoletes some of the classification characteristics from [122]. In particular the *Programming Model* and *Reconfiguration Mode* issues seem to no longer be appropriate classification characteristics.

In [122], *Programming Model* distinguished between application-specific and instance specific SAT solvers. Skliarova and Ferrari identified a trend that the more recent solvers leaned towards application-specific architectures, and this trend has held true. All of the works surveyed here are application-specific solutions, and all of them directly addressed instance specific designs as taking far too long to implement the instance circuit. Since 2004, only one publication proposed an instance-specific design[69]. This work was published as an architectural exploration that mapped the target CNF file to a VHDL package structured similarly to a database such that constant propogation results in a highly optimized clause evaluation circuit. While interesting as an architectural exploration, this solver isn't addressed in detail here because performance is not competitive with software even when compilation time is not considered, only showing speedup on a handful on unsatisfiable problems.

Similarly the works discussed in [123] could be classified according to *Reconfiguration mode*. Solvers could use static configuration or dynamic reconfiguration for different steps of a specific problem instance. All of the approaches surveyed here program fixed logic structures onto the FPGA and use memories, either internal block RAMs or external DRAM memories to store the instance data. Even the approaches such as [41, 58] that modify internal circuit representations in memory only do so with the regular BRAM write ports. None of the approaches considered actually change the physical configuration of the FPGA. By Skliarova's

Table 2.2: Scaling Complexity of various Hardware Solvers

Solver	Capacity Scaling	Accelerated Portion Scaling
Safar et al [111, 112]	Quadratic	Quadratic(Memory), Linear(Logic)
Gulati et al[57, 58]	Linear	Quadratic(Logic)
Haller and Singh [59]	Linear	Fixed(Logic) - Two BCP cores
Davis et al [41, 42]	Linear	Linear(Memory)
Thong et al [128, 129]	Linear	Linear(Memory)
Yuan et al [134]	Linear	Exponential(Logic)
Kanazawa, Maruyama[72, 73]	Linear	Fixed(Logic)

definition, it is proposed that all the works here would be considered statically configured.

Scalability

In order to assess how well the studied works would scale on modern hardware, a series of projections were made based on published information and educated assumptions. Two separate aspects of scaling were assessed. One was how well a solution could scale to solve larger instances on a modern FPGA, and the second is how well the hardware accelerated portion of the algorithm scales. This second metric is important for solutions that partition the problem into sequentially processed bins[58, 134]. An abstract view of these estimates is seen in Table 2.2.

Scaling estimates are based on the device utilization, scaling formulas, and architectural descriptions provided in the various publications. Due to variations in the available information, some projections will be more accurate than others. In general, the projections are purposefully optimistic and aim to present the most fair comparison possible based on which type of resource will limit instance capacity or accelerator size, which can be limited by logic resources or internal/external memory space.

While outside the scope of this investigation, it is important to consider scalability within the context of software solutions as well. Software solver capacity can scale to support most tractable application instances due to access to plentiful DRAM. Software performance in terms of BCP throughput, however, was shown in [128] to drop faster than Thong's solution as problem size or the number of threads increased. This was attributed to Thong's co-processor relieving pressure on the memory subsystem as larger instances result in more cache evictions and multiple threads increased contention for limited memory bandwidth and cache residency.

Scaling to solve larger problems

Davis et al. [42] supported 256 parallel inference engines on a XC5VLX110T device with 296 BRAMs. Each engine used one BRAM block consuming 256/296 18Kb memories overall. Since their inference engines are of fixed size, it is reasonable to assume roughly linear scaling of BRAM usage. Their design is primarily limited by the number of BRAMs, using 91% of available BRAMs, but less than 20% of the LUTs on the XCV5LX110T. Given the improvements in the tools and manufacturing it may be reasonable to assume that 4096 parallel inference cores running at the same 200MHz rate would be possible, using 95% of the Xilinx XCKU115 BRAMs.

The maximum problem size supported by Gulati[58] should also scale roughly linearly according to memory resources, but it isn't clear what the memory requirements of their bin encoding is.

Safar's approach [111] evaluated all clauses in parallel, and could support up to 511 clauses on a Xilinx XC2VP30 device. In [111] capacity was limited by the logic consumed by the parallel clause evaluators. Each clause evaluator is of fixed size and logic utilization should scale linearly with instance size. Memory usage on the other hand scales quadratically, and quickly becomes the limiting factor for this approach. The Memory VEOC component of their design requires two bits of storage per clause for each variable. In [111], 65 total 512x32 memory blocks were used, with 32 used for the VEOC, 32 presumably for the ClauseDB, and a single BRAM is used for the register file. Memory requirements for this design should be approximately $4C^2$ bits where C is the number of clauses. If two XCKU115 BRAMs are reserved for the register file, then the maximum instance size can be expressed as $V, C = \sqrt{(4318 * 18 * 1024)/4}$, which means that approximately 4460 variables/clauses that could be supported. Supporting the required 8920 $(4460 * 2)$ bit read width is not an issue, and the roughly 8X linear scaling of the clause evaluator logic would use approximately 27% of the Xilinx XCKU115 device.

No attempt was made to scale Thong's approach as it was already implemented on a modern Stratix V FPGA with 40 Mbits of internal memory, and his compression techniques are highly instance-specific relying on memory partitioning, global address minimization, and enhanced BCP to fit larger circuits.

Additionally several approaches [59, 73, 134] stored the clause database in DRAM and requested data as needed. The maximum problem size supported by these approaches should scale gracefully with DRAM capacity. No attempt was made to scale these approaches to a larger memory, as the largest published instances in [59, 73] are already sufficient for most currently solvable real-world instances; it's expected that [134] would also be able to store such large instances provided sufficient DRAM.

Figure 2.3 compares the maximum supported problem size for the considered solvers. Where possible, it includes scaling projections to a Xilinx XCKU115 part. Due to the large variance between solvers, it was necessary to plot this on a logarithmic scale. In terms of maximum

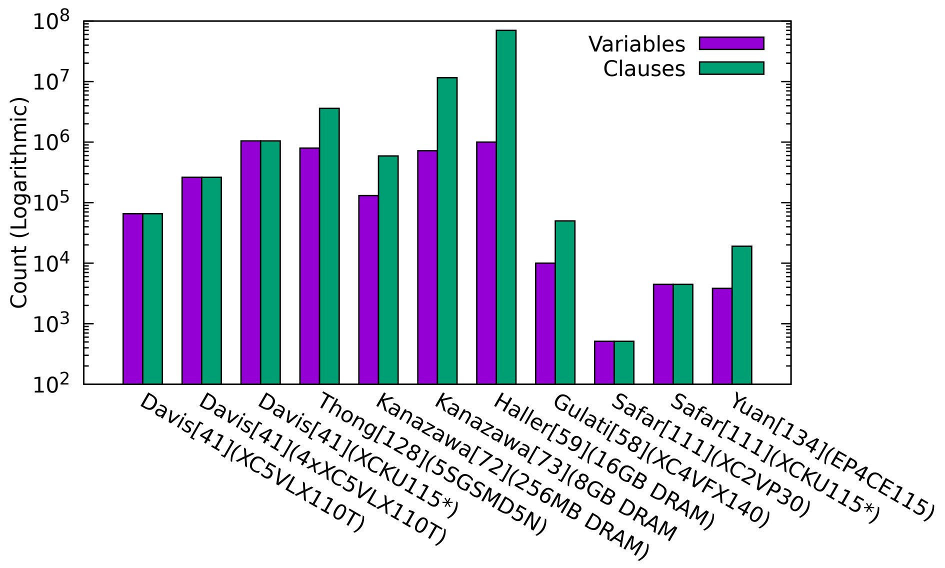


Figure 2.3: SAT Solver supported Problem Size
(Projections marked with *)

number of clauses, the DRAM-based approaches of [59] and [73] have a clear advantage supporting 70 million and 11.6 million clauses respectively. However, [128] (795,369) and [41] (1M, projected on the XCKU115) are competitive in terms of number of variables supported, even while using internal memory for storage. All of the aforementioned approaches are sufficient to solve many real-world instances, though the 1M clause limit of [41] limits applicability to some larger problems. It's also likely that [134] and [58] would scale to support much larger problems, but there is not enough published information to confirm this. The 4460 clause limit of [111] greatly reduces applicability to real-world problems.

Scalability of Accelerated Portion

The maximum supported problem size doesn't tell the whole story. Some of the presented approaches partition the problem into bins such that intra-bin resolution is performed with parallel operations, but the bins are processed sequentially. In these approaches, if the bin size does not scale along with the maximum problem size, then the solver will be unbalanced where time spent sequentially processing bins will outweigh the speedup gained via accelerated processing of each bin. Calculations were thus carried out in order to isolate the accelerated portion of each solver and predict how well they would scale to a larger Xilinx XCKU115 FPGA.

In Gulati-2009, the authors support relatively large instances on a Xilinx XCV4FX140 device (up to 10K variables and 280K clauses); however, the accelerated portion supports a bin size of 75 variables and 50 clauses. These bins are swapped sequentially by the decision engine. The parallel solver portion scales as a function of $LUTs = 20 * V * C + 300 * V$[58], and they calculated that ideally $C = \frac{2}{3}V$. Solving the resulting quadratic equation we find that a KU115 should be able to support a 285 variable and 190 clause partition with fully parallel propagation and conflict resolution. This offers only a 3.8X increase from the projected 75 variable and 50 clause partition size on the XC4VFX140 device[58] despite a full order of magnitude increase in on-chip resources.

Yuan et al.[134] brute-force all combinations of a fixed number of variables in parallel. Thus, the accelerated portion of their solution scales exponentially at the rate 2^V with V being the cutoff for the number of undecided variables where control is switched from software to hardware. A 13 variable brute-force circuit was implemented on a 114,480 logic element Terasic DE2-115 board. If the number of logic elements is divided by 2^{13}, then at most roughly 14 LEs are needed per assignment possibility. The Xilinx XCKU115 device claims 1,451,100 LEs, and thus could potentially support processing 103,650 assignments in parallel. It is unlikely then that even the largest FPGA would be able to process more than 17 variables in parallel. It's unlikely that performance would scale with problem size as the ratio of total variables to sub-problem variables grows.

Figure 2.4 shows the relative (logarithmic) scaling of accelerated portions of the various designs going from the original device to the XCKU115. This is an approximation as the effectiveness of the exploited parallelism is instance-specific for both [41] and [58]. The efficacy of both of these accelerators are highly dependent on how effectively an instance can be partitioned and mapped to hardware.

Kanazawa's[73] solution is not displayed here as the level of parallelism is fixed by his pipeline length, which is dependant on external memory access times instead of available logic or local memory. Kanazawa[72] only has 20 clause evaluation cores, but based on his analysis of clause to variable ratios in 3-SAT problems, these are sufficient to evaluate all possible implications of a single decision within a 3-SAT circuit. In fact, the 12-stage pipeline described in [72] is capable of doing this for four concurrent threads.

The approach taken in[59] has fixed levels of parallelism based on the two-watched literal scheme but performance is limited by memory throughput and any performance scaling would be due to the upgrade to DDR4 memory on modern platforms. It's important to note that software would benefit equally from this transition in addition to increased amounts of L3 cache (up to 30MB) compared to the contemporary CPUs.

Thong's approach in [128, 129] would benefit directly from increases in BRAM capacity either by supporting larger instances with the same number of threads, or by allowing even more threads since his approach stores redundant variable assignments for each thread.

2.3.4 Generalized Insights

While the hardware solvers discussed above took many different approaches, each had a set of insights and contributions that should be taken into account by anyone attempting to create a new reconfigurable SAT solver architecture.

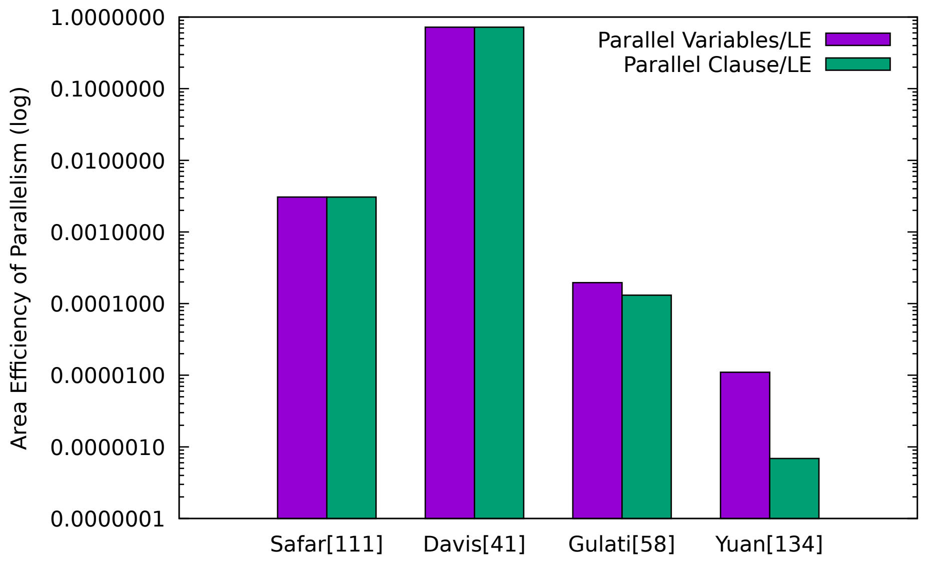


Figure 2.4: Scaling of Parallel Portion

Implication Time must be Bounded

As Davis [41, 42] points out, software SAT solvers consistently perform BCP operations on multi-GHz CPUs within thousands of clock cycles. In order to be competitive, a hardware accelerator must keep implication time bounded to tens of clock cycles. Performance *cannot* scale linearly with problem size when competing with the software two-watched literal scheme which effectively bounds BCP time. Implication time can be bounded by scaling in terms of area usage rather than execution time by extracting more parallelism. This is the approach taken in [41, 42, 111] where all clauses are processed in parallel and solving larger problems requires more procesing elements. Gulati et al [58] have a consistent implication, and conflict resolution time, but only within a single bin.

In [128, 129], there is no explicit bound on implication time, but there is an implicit bound based on the maximum number of occurences of a variable across clauses. While individual implication time is longer than [41] due to sequential clause processing, overall throughput is higher when multi-threading is utilized[128].

Solutions Must Scale to Solve Real Problems

In order for hardware accelerated SAT solvers to be relevant, they should be able to support larger problems that are actually useful. Problems in the domain of AI planning, software verification, and EDA tend can have up to millions of variables and clauses. Some of the approaches presented here only support smaller problems with poor ability to scale to larger ones.

Some approaches such as [111] are limited in terms of maximum supported instance size, and it is unlikely that such a solver would be very practically applicable. Other approaches, such as [58, 134] should support large instances but it isn't clear that performance would scale well with instance size due to the relatively small bins that are being accelerated in parallel. Ultimate [59, 73, 128, 129] seem to be the only solvers that can scale to support larger real-world instances while still exhibiting a clear speedup.

Importance of Partitioning Methods

Many of the published hardware architectures used partitioning methods in some way or another. Approaches that use generic cores to process clauses need to partition a SAT instance into smaller pieces that can be solved in bounded time by that core. In [41, 42], partitioning was used to allocate groups of clauses to each implication engine such that the maximum number of clauses are evaluated in parallel, and that each clause engine would produce at most a single implication by ensuring that unique literals do not occur multiple times in the group. [128, 129] used partitioning methods to group clauses together and maximize local addressing; thus increasing storage efficiency.

Partitioning is also used to allow acceleration of large circuit instances to a relatively small FPGA fabric. This approach is used in [58] where the circuit is partitioned into small bins that can each be solved in near constant time. Even in [57] where it seems an infinitely large ASIC fabric is assumed, clauses are partitioned using `hMetis` into banks as a workaround for the quadratic resource scaling of their core solver.

Safar et al.[111] make no attempt at partitioning the problem into solveable sub-problems and problems that do not fit on the FPGA cannot be solved. Furthermore the quadratic memory scaling of this approach as discussed in 2.3.3 means that simply buying a larger FPGA would not be a reasonable solution either. On the other hand, their core solver is more area efficient than [58] and it would be interesting to combine Safar's core solver with Gulati's partition based approach.

Multi-Threading on FPGA Allows Greater Software Parallelism

While the approaches presented have primarily focused on accelerating SAT by speeding up each iteration of solving a single SAT instance, there is a limited amount of parallelism available in each iteration. At first glance, it would seem that every clause within a BCP process can be evaluated in parallel, but the need to account for implications complicates this. In [41] related clauses were partitioned into different groups so that each inference engine only produced one new implication per iteration, and these implications needed to be serialized and checked for conflicts after each iteration. In [128, 129] it was noted that the implication dependency chains in BCP made it an inherently sequential process. Instead of processing clauses in parallel for each propagation, Thong supported multi-threading to maximize throughput, utilization of processing elements, and mask communication latency.

In [128], there is a thorough analysis of how performance is affected by the number of threads and the problem size for both software solvers and his hardware-assisted solver. Generally as problem size increases the performance benefit of adding threads drops off until adding threads actually reduces performance. This is attributed primarily to limited memory bandwidth and cache thrashing on the CPU. This drop off is delayed or mitigated when using the BCP accelerator; in fact for the tested benchmarks using the maximum 12 threads is always faster. Intuitively, moving BCP for each thread into the FPGA relieves pressure on the memory subsystem of the CPU allowing more effective cohabitation of larger numbers of threads.

Kanazawa and Maruyama[71, 72, 73] implemented a multi-threaded WalkSAT algorithm. Similar to a portfolio approach, WalkSAT threads can operate independently of each other without communication making their architecture ideal for multi-threading. Performance seemed to scale linearly with the number of threads.

Both of these approaches illustrate how multithreading can be leveraged to exhibit more parallelism and more efficient FPGA utilization than is natively exposed in the SAT problem.

2.4 The Path Forward

Thus far, the current state of hardware and software SAT solvers has been addressed, as well as the challenges posed when developing an effective reconfigurable SAT solver. Some potential strategies and technologies that can help overcome these challenges are examined here.

2.4.1 Path Forward for Hardware Acceleration

Accelerate non-BCP portions

Some of the works proposed by [41, 42, 128, 129] accelerated BCP until overall speedup neared the theoretical limits implied by Amdahl's law, it is unlikely to be fruitful for future works to pursue pure acceleration of BCP. New methods and algorithms that make decisions, resolve conflicts, and bactrack hardware may be necessary to achieve significant advances.

Tailor Designs for Hardware Solutions

It is tempting directly incorporate the impressively successful software techniques in hardware solvers. However, this approach often leads to inefficient hardware designs that do not scale well with the size of the problem. Effective software techniques such as CDCL style conflict resolution and clause learning do not efficiently map to hardware. One example is the difficulty in scaling [58] to larger devices. [111] presented an area efficient implementation of non-chronological backjumping and learning in hardware. Algorithmic trade-offs resulted in some loss of performance, but allowed for an area efficient implementation of high-throughput clause evaluators, backjumping logic, and a learning mechanism. This approach had limitations in terms of memory scaling, but it did show the potential gains of hardware-centric solver algorithm. Perhaps less successful software approaches such look-ahead or non-CDCL/DPLL based algorithms could be re-considered to better leverage the massively parallel heterogeneous resources available in modern FPGAs.

Embrace Emerging Technologies

One of the recurring themes in many of the papers discussed here has been instance size limited by BRAM capacity[41, 42, 58, 128, 129, 134] or solvers with performance limited due to the use of off-chip memory [59, 72, 73]. Emerging memory technologies such as Hybrid Memory Cubes (HMC) and High-Bandwidth Memory (HBM) offer up to 20X the bandwidth of DDR4 based solutions[131]. This also is a unique advantage for hardware accelerators, FPGAs with integrated HBM will be released soon[131], and have been available with HMC support for several years, but general purpose processors are stuck with DDR4 for the foreseeable future.

Works that presented hardware/software solutions such as [41] or [58] found that a significant portion of execution time was taken up by communication with the processor. Newer designs could take advantage of more tightly coupled platforms such as Xilinx's UltraScale Zynq and Intel/Altera's HARP platform offer a significant opportunity for a hardware/software partitioned SAT solver that takes advantage of high-bandwidth, low-latency links between CPU and FPGA. Hybrid SLS/CDCL solvers could make a lot more sense in embedded FPGA applications such as [130] where parallel FPGA resources could compensate for lower-performance CPU cores running CDCL algorithms.

Build True Application-Specific Designs

One thing that has been made clear in SAT competition results [3] is that different solver algorithms and heuristics excel at solving different instance types. While CDCL solvers mostly dominated the large application benchmarks, many tracks benefited from solvers that perform SLS either before or concurrently with CDCL search. In addition to the difference between different problem types, there is significant difference between SAT and

UNSAT instances and how they are best solved regarding learned clause management, decision heuristics, and restart strategies[100].

In the literature, an *application specific design* is one where generic instances can be loaded onto hardware without requiring recompilation. This can be counter-intuitive as one may expect an *application specific* solver to be optimized for a specific class of applications. The majority of the surveyed works benchmarked against a variety of instances that fit on their FPGA and reported overall speedups. Given the level of attention paid to tuning software heuristics and parameters to specific types of problems, it is unlikely that creating a general-purpose hardware accelerator is possible. Instead, hardware designers should carve a niche within a problem domain (cryptography, AI planning, EDA, etc.), instance type (application, hard combinatorial, random), instance size/difficulty (small/difficult, large/easy), and SAT vs. UNSAT and build a design optimized to excel at solving that problem. If not planned from the beginning, when proposing an application-specific architecture, researchers should profile their solution and figure out what kind of instances are best suited to their platform.

2.4.2 Expand Acceleration to MaxSAT, SMT, and QBF

Boolean SAT solver algorithms and implementations have been developed to maturity over a half-century to a point where software solvers are extremely performant. This has led to SAT solvers being used in scenarios where other forms of theorems such as Maximum Satisfiability (MaxSAT), Satisfiability Modulo Theories (SMT), or Quantified Boolean Fields (QBF) are more natural and efficiently mapped to the application at hand. There are many such scenarios where less efficient mappings to SAT are used because SAT solvers are mature enough to overcome inefficiencies and still perform better.

This leaves a potential opening where new approaches to accelerate solvers of more complex theorems such as MaxSAT, SMT, or QBF could make a more substantial impact in a less mature field.

In addition, MaxSAT and its variants are NP-hard, and could potentially offer more opportunities for parallelization.

2.5 Summary

The Boolean Satisfiability problem is a theoretically significant problem with compelling real-world applications. The importance of SAT has resulted in a large volume of work attempting to quickly resolve SAT instances including a highly-efficient sequential software algorithms and implementations, parallel solving approaches for shared and distributed memory systems, as well as hardware accelerators targeting reconfigurable hardware.

The most widely used and performant SAT solvers are complete solvers based off of the CDCL algorithm. There has been some success accelerating these solvers using parallel portfolio approaches, but these approaches are not very scalable as it is difficult to prevent worker threads from doing redundant work. Even with sophisticated information sharing schemes that minimize communication overhead, it is likely that there are inherent limitations to concurrency based on the depth of the resolution proof[75].

A less explored, yet promising method for extracting scalable concurrency from SAT then is that of massively parallel local search which has shown good speedup on select instance classes in [7].

Parallel solvers for MaxSAT and WPMS have also been developed with various levels of success. While efficient parallel implementations of portfolio and search-space splitting algorithms show performance gains, none as of yet have demonstrated consistent scalability past 8-16 cores on parallel machines.

Since the publication of [122] in 2004 multiple published works have advanced the state of application specific hardware SAT solvers. At the same time, software solvers advanced rapidly with the release of GRASP[87], Chaff[96], MiniSAT[46], SATZilla, and others advancing software solvers to the point where instances with millions of variables and clauses are capable of being solved in minutes on commodity hardware. Hardware accelerators were thus faced with a moving target where software performance increased much faster than CPU manufacturing and clock speeds, with progress being aided by competitive yearly SAT competitions. Keeping pace and exhibiting speedup over such sophisticated software solvers is a daunting task and the various works surveyed here took different approaches. Gulati et al [57, 58] implemented the GRASP engine in a clever hardware design that allowed for parallel constraint propagation and conflict resolution. While this is the only approach capable of parallel conflict resolution and learning, the core solver scales quadratically. This requires partitioning instances into smaller, sequentially solved bins, thus limiting performance on larger or more difficult problems. Davis et al and Thong et al [41, 42, 128, 129] avoided competing with software solvers and instead offered BCP coprocessors capable of accelerating BCP for any DPLL based solver. These approaches were successful and scalable with up to 5-6X speedup. Overall, the performance benefits shown by these approaches are likely not enough to justify the expense and complexity of using them in real-world applications.

Kanazawa and Maruyama implemented WalkSAT, a SLS solver in hardware. WalkSAT maps well to hardware and their design exhibited good performance and scalability. However, performance was limited by off-chip memory latency, and WalkSAT in general does not perform very well on industrial instances. They also implemented a hardware implementation of the Dist WPMS algorithm to solve the special case of Weighted Max2SAT and achieved 7X speedup over the software implementation.

This chapter introduced the SAT problem and discussed the background of popular techniques and algorithms employed by modern solvers. A survey of hardware accelerated SAT solvers was performed with a particular focus on works published after the survey presented in [122]. Hardware solvers were classified and analyzed with regards to performance and scalability. An attempt was made to extrapolate these parameters beyond the original publications to see how various approaches would map to a modern FPGA. Higher-level analysis and consideration was also taken to contrast the state of the art in hardware solvers with that of software solvers. The various constraints and requirements for a successful hardware solver was discussed along with a proposed path forward.

Attaining significant acceleration with reconfigurable hardware or scalably parallel software is a challenging problem. Sequential solver performance progressed at breakneck speeds with the rapid-fire improvements in performance from GRASP[87], Chaff[96], and MiniSAT[46]. However, this field has matured and most current solvers follow the MiniSAT playbook if they haven't directly built on the MiniSAT codebase. Recent research in SAT has focused on tuning heuristics for different instance classes[100] and improving SLS search[12, 14, 15, 16], portfolio[132], and parallel performance[7, 8, 18, 61, 63, 64]. Much progress has been made recently in directly solving generalizations of SAT such as MaxSAT[30, 80, 81, 127] or Sat Modulo Theories[51, 98], that may have traditionally relied on the rapid performance improvements in underlying SAT solvers. As the advancement in core sequential SAT algorithms seem to have stabilized, an opportunity emerges to create a concurrent, or hardware accelerated SAT solver that truly advances the state of the art. The lessons learned from the papers discussed above motivate the work presented in the ensuing chapters.

While [7] showed the potential of massively parallel local search, [73] proved that SLS search could be efficiently implemented in FPGAs. Thus, in Chapter 3 a Massively Parallel SLS solver is presented that supports operation on compute clusters as well as on FPGAs. Since neither work put much thought into deliberate seeding of the search cores, this is investigated in Chapter 4.

Since many algorithms require the ability to prove unsatisfiabllity, a hybrid solver is presented in Chapter 5 that extracts parallelism through distributed SLS search threads, but achieves completeness as unsatisfied clauses are gathered from the search threads and passed to a CDCL solver thread asynchronously.

Finally, the strong performance of simple SLS algorithms for MaxSAT and WPMS on industrial and crafted instances motivated the work presented in Chapter 6 that applies massively parallel local search to WPMS, and the application of this WPMS solver to the probabilistic inference domain is presented in Chapter 7.

Chapter 3

Accelerating Satisfiability with Parallel Local Search

Following from the analysis of the current state of concurrent and hardware accelerated satisfiability presented in Chapter 2, this chapter presents an effective concurrent SAT solver that can be efficiently implemented on reconfigurable hardware or compute clusters.

The DPLL and CDCL algorithms that have been dominating software SAT solver competitions are *complete* algorithms that will eventually finish execution and resolve an instance as being either satisfiable or unsatisfiable. On the other hand, *incomplete* solvers search for a satisfying solution without any guarantee of ever finding one or being able to prove unsatisfiability. Incomplete solvers predominantly employ *Stochastic Local Search* (SLS) algorithms, greedy, probabilistic algorithms that treat SAT as an optimization problem to maximize the number of satisfied clauses with each flip of a variable assignment. Popular early SLS algorithms include Greedy SAT (GSAT)[119] and the follow-up Random Walk SAT (WalkSAT)[118]. Modern SLS solvers such as Sparrow and probSAT use precalculated probability distributions to simplify and speed up the process of evaluating variables to flip. Efficient decision making allows SLS solvers to make tens of millions of flips per second; they excel at problems with no easily discernible structure where clause-learning and intelligent backjumping is actually a detriment. Additionally, these solvers are trivially parallelized as solver instances with different seeds are fully independent. The parallel version of probSAT (pprobSAT)[16] won the Parallel Random SAT track of the 2014 SAT Competition [3]. In [8], near linear speedup was shown on up to 512 cores on selected instance classes.

The approaches to FPGA acceleration and concurrent SAT solving discussed in Chapter 2 exhibited varying levels of success, but have seemingly failed to gain widespread usage. Most of them accelerated DPLL/CDCL algorithms. Fewer attempts have been made to implement SLS solvers in hardware despite the seemingly natural mapping of an embarrassingly parallel scaling ability and simplified control flow.

In this chapter, a parallel local search solver for SAT is presented. The general approach of distributed SLS based on the probSAT[15] algorithm is presented both as an accelerator targeting reconfigurable hardware as well as an OpenMP implementation that can run on general purpose compute clusters. The FPGA implementation is a High Level Synthesis (HLS) adaptation of the open-source C implementation of probSAT[13], and the OpenMP version encapsulates the core probSAT functionality as a C++ class that can be operated upon by worker threads. Pursuit of this implementation was motivated by the strong performance of probSAT in recent SAT competitions, along with a minimal C implementation that seemed to lend itself to HLS. Instead of trying to compete with a modern x86 processor in terms of sequential performance, the goal was to create a minimal solver core that could perform within an order of magnitude in terms of flips/second while still being small enough to be replicated many times on a single FPGA to exploit the embarrassingly parallel nature of solver instances spawned with different seed values. Another goal was to have the solver core load instance data from memory, allowing for an application-specific implementation style where cores could be fixed in the FPGA and new SAT problems could be loaded into memory while still exhibiting a speedup over contemporary software solutions.

The proposed solver meets the above criteria and exhibits strong performance on a set of small, difficult SAT problems with speedups of up to 828x and 99x when compared to

MiniSAT and probSAT respectively.

3.1 Related Work

Many attempts to accelerate SAT solvers are found in the literature. An excellent survey of reconfigurable hardware SAT solvers was published in 2004 by Skliarova and Ferrari[122]. This survey includes some notable contributions, but most are hamstrung by the limited FPGA devices of the time. Many early solvers were *instance specific* meaning that a hardware accelerator was crafted for a single SAT instance. These designs showed significant speedups, but only if FPGA implementation runtime was ignored. Skliarova and later authors noted that for a SAT accelerator be effective, it was necessary for *application specific* designs that allowed arbitrary instances to be solved on a general purpose SAT solver design, otherwise the complexity of the SAT problem was simply being pushed into the EDA tools[129]. It is difficult to tell how the older designs would perform on modern devices as there is no source or even binary availability and many of the presented benchmarks are trivial by today's standards if they are still available.

A brief survey and analysis of Hardware SAT solvers published since 2004 is provided in Chapter 2 with a more thorough survey presented in [125].

The most relevant works are a series of WalkSAT implementations from Kanazawa and Maruyama [71, 72, 73] published between 2005 to 2014 with incremental improvements. WalkSAT is an incomplete algorithm that performs a random walk by choosing an arbitrary unsatisfied clause and probabilistically flipping variable assignments to minimize the number of unsatisfied clauses. WalkSAT is particularly well suited to implementation in hardware as the control structure is much simpler than CDCL; there is no learning, and multiple threads can operate fully independently. Since algorithm execution time is not deterministic, WalkSAT performance is typically measured in terms of flips/second. This is a generally reasonable approach, but sometimes better heuristics are capable of solving the problem with significantly less flips. In [71] a 10-stage pipelined circuit was presented to evaluate clauses, select an unsatisfied clause, flip a random variable within that clause, and repeat the cycle until the instance is satisfied. In order to saturate the pipeline, they process four independent threads at a time and found that performance scaled linearly with the number of threads. In [72] the design was enhanced to solve larger problems. Whereas the previous design evaluated all clauses in parallel, this one only evaluates clauses containing the negation of a literal that is a candidate to be flipped. In all the attempted benchmarks, the maximum number of clauses containing a single literal was 20 and on average would be $N_c \times 3/N_v/2$ for 3-SAT problems. Thus, this design only required 20 parallel clause evaluators for the main stage compared to [71] where a clause evaluator was instantiated for each clause in the instance. This design was also pipelined and supported up to four threads. [72] also added support for off-chip storage which predictably improved circuit capacity and lowered performance due to the higher capacity and latency of DRAM vs on-chip memory. Finally in [73], the same authors present a further refinement of the design with heuristic improvements and native support for k-SAT instances along with a variable-way cache to hide the DRAM access latency. A set of impressively sized benchmarks were run on a hardware simulator with up to 663574 variables and 15489863 clauses boasting speedups of 1.16-8.07X compared to an Intel Core i7-3770.

In terms of software implementations, [8] presented experimental results where they found that unmodified SLS search threads scaled well up to 512 cores on crafted and verification instances, but not so well on random and quasi-group with holes instances. Interestingly, their previous work in [7] showed that using the same algorithms with a sophisticated information sharing mechanism did not scale very well.

The motivation for this work is that implementing a full hardware CDCL solver modeled after software solutions is not efficient. As shown in [58], it is possible to create a fully parallel CDCL engine, but what is lacking is a solver that incorporates dynamic variable selection, propagation, conflict-analysis, and clause-learning in a way that is area-efficient. Using Gulati's scaling calculations one could only support 300 clauses per bin even on a modern Xilinx Ultrascale XCKU115 part. The need to partition into smaller bins limits general applicability to small instances or ones that partition cleanly. BCP accelerators such as [41, 42, 128, 129] are limited to 5-10X speedup, which may make it difficult to justify the cost of an FPGA solution. The WalkSAT approach in [73] is a natural fit for hardware implementation but speedup is limited due to the low performance of DRAM and the limited number of threads. Additionally, there are newer SLS algorithms with potential to outperform WalkSAT that have been successfully competing in SAT competitions either as standalone applications or in a hybrid/portfolio style solver.

The probSAT solver in particular seems to be a natural fit for hardware implementation as it claims to be the "one of the simplest solvers ever presented"[15] and the parallel version won the Random 3-SAT track in the 2014 SAT Competition.

3.2 probSAT Algorithm

As proposed in [15] probSAT is presented as a new class of SLS solvers. Most SLS solvers calculate a score based on the *make* and *break* values of flipping a particular variable's assignment. Often the score is as simple as $make - break$. Only if no score improvement is possible (as in the case of local minima) is a decision made probabilistically. The probSAT solver is inspired by the Sparrow solver, which makes its probabilistic decisions based upon a probability distribution instead of a decision procedure. Where the probSAT class of solvers differ from Sparrow and other SLS solvers is that probSAT solvers *only* use probability distributions in their search, there is no direct score based decision procedure.

In [15] the authors experiment with different variations of the probSAT concept to determine which makes the most effective SAT solver. The probability distribution for selecting the next flip variable is solely based on the *make* and *break* values. The authors experimented with different weightings of these values as well as exponential and polynomial probability distribution functions. Interestingly, they found the best results when the make value was weight to zero or near zero meaning one could ignore the make score altogether without a significant impact on performance. They also found that for 3-SAT instances the polynomial distribution function worked best and for 5-SAT instances the exponential distribution performed better. This polynomial break only solver was even able to solve all 180 instances of the `3satExtreme` benchmark set where previously only survey propagation was competitive. The algorithm also exhibited linear scaling with problem size as the number of flips required per variable remained relatively constant.

The lessons learned in [15] were applied successfully to optimized implementations that won the sequential Random SAT track at the 2013 SAT competition and the Parallel Random SAT Track at the 2014 SAT Competition[3]. The parallel pprobSAT variant simply spawned multiple jobs on separate CPU cores with different seeds and parameters. The hardware implementation presented here is based on this concept. The basic idea is that since the probSAT algorithm is so simple that an FPGA implementation could hold several hundred cores with the potential to increase the overall flips per second by orders of magnitude on a single large FPGA. Furthermore, since probSAT is memory based with a simple memory layout, such an FPGA architecture could be designed to be application specific with a fixed number of independent cores that read the clauses from memory.

For 3-SAT problems, probSAT implements the algorithm shown in Algorithm 1.

Algorithm 1 probSAT Algorithm for 3-SAT[15]

```
Input: Formula $F, maxTries, maxFlips$
Output: Satisfying assignment $\mathbf{a}$ or UNKNOWN
 1: for $i = 1$ to $maxTries$ do
 2:     $\mathbf{a} \leftarrow$ Randomly Generated assignment
 3:     $falseClauses \leftarrow$ Initial set of unsatisfied clauses
 4:     for $j = 1$ to $maxFlips$ do
 5:         if $(falseClauses = \emptyset)$ then
 6:             return $\mathbf{a}$
 7:         end if
 8:         $C_u \leftarrow$ Randomly selected unsat clause
 9:         for $x$ in $C_u$ do
10:             compute $breaks$ of flipping $x$
11:             Look up $f(x, a)$ using $breaks$
12:         end for
13:         $var \leftarrow$ random variable $x$ according to probability $\frac{f(x,a)}{\sum_{z \in C_u} f(z,a)}$
14:         flip($var$)
15:         update $falseClauses$ according to new $\mathbf{a}$
16:     end for
17: end forreturn $\mathbf{a}$
```

3.3 HLS Implementation of probSAT

One of the motivating factors for choosing probSAT was the simplicity of the algorithm. In fact the version of the code posted on GitHub[13] contains around 750 lines of documented C

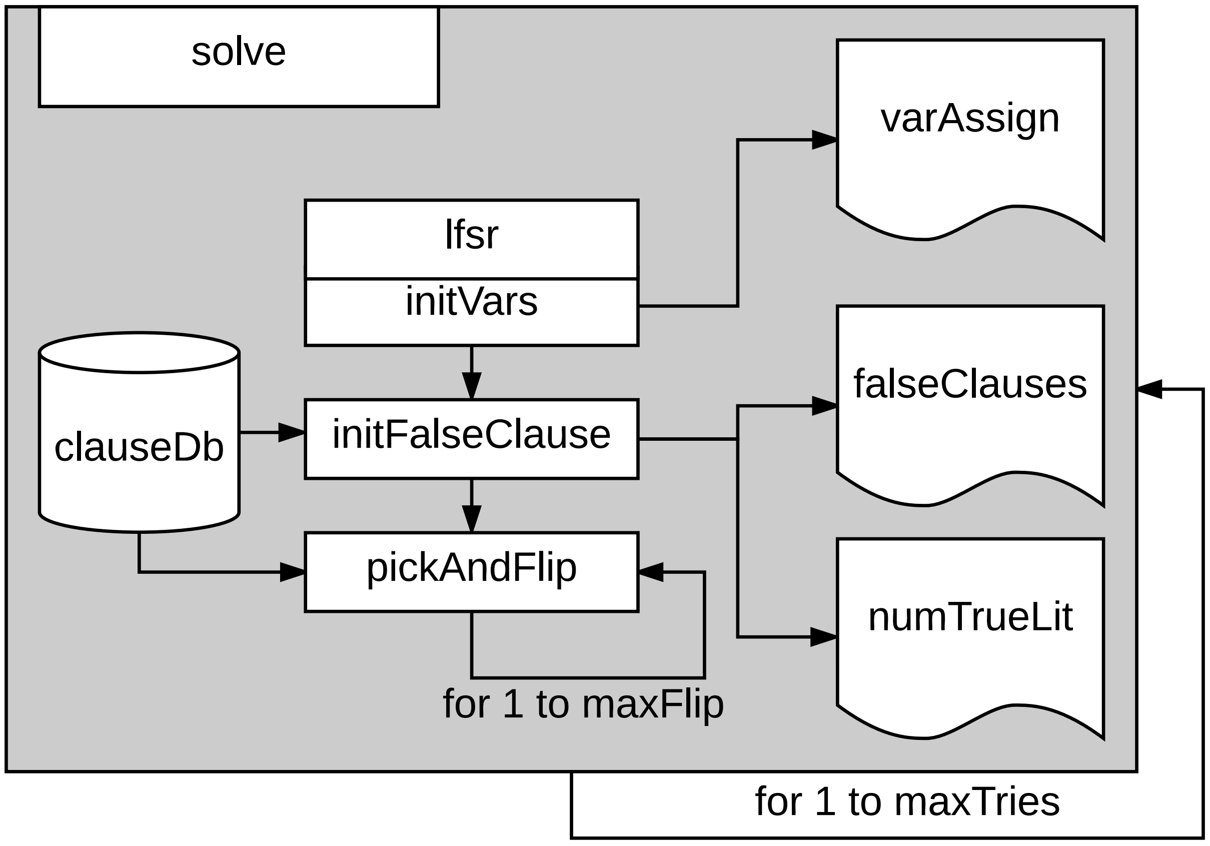


Figure 3.1: Solver Block Diagram

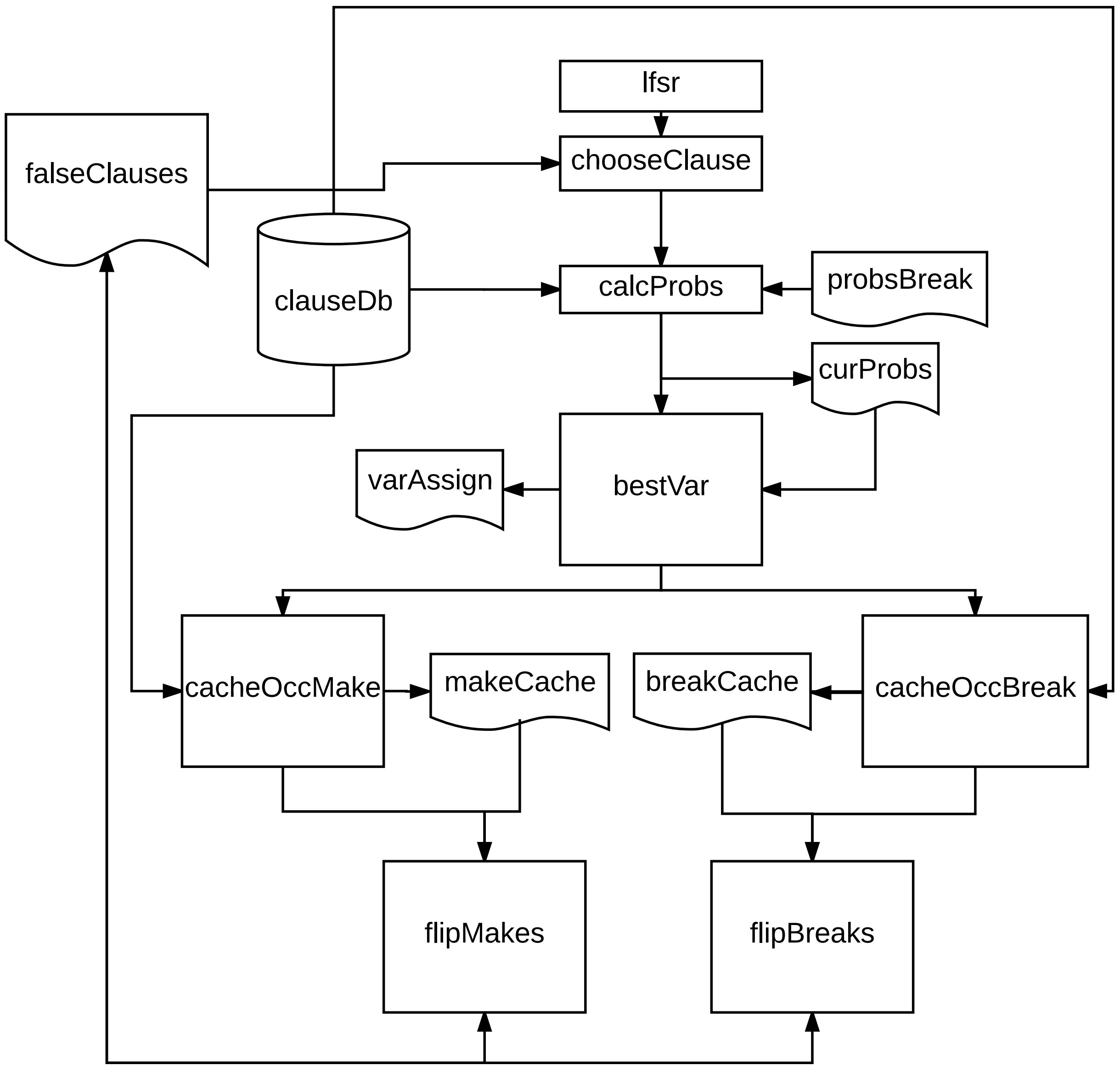


Figure 3.2: PickAndFlip Module Block Diagram

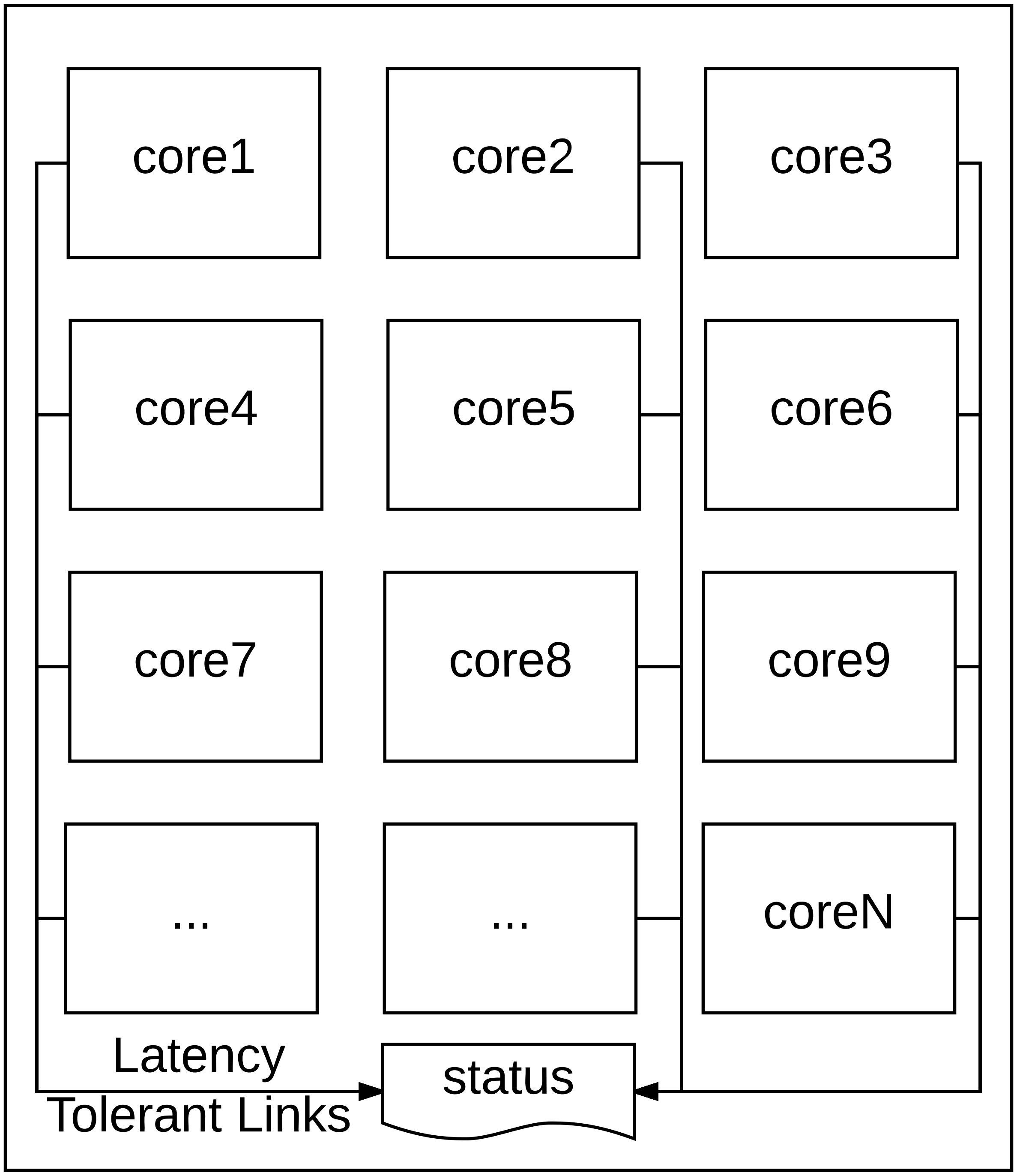


Figure 3.3: Many-Core Solver Implementation

code. The simplicity of the code and pre-allocation of all memory structures also implied a straightforward translation to HLS. The actual process ended up being a bit more involved.

First, the core solver functionality was separated into a callable function that would serve as an HLS top-level function. In this iteration, the original data structures were preserved as they were simple pre-allocated array-based lookups that would map well to internal BRAMs. The *pickAndFlip* function was first isolated as an HLS top-level, and once that function was working, a higher-level `solve` function was made into the top-level function. The *solve* function simply wrapped the *pickAndFlip* function with a `for` loop that performed up to $maxFlips$ flips before giving up. The default probSAT implementation allocated two-dimensional arrays with null terminated arbitrary length rows to account for instances where clause length and occurrence list length for each variable is different. As such `while` loops were used to iterate over the databases and terminate when null values were encountered to indicate the end of a row. These `while` loops with non-deterministic loop counts were replaced with bounded for loops and the clause database structures were modified to be full $n*m$ matrices. This results in more efficient hardware implementation and greatly simplified performance profiling and optimization. Being a probabilistic algorithm, probSAT heavily uses the `rand()` function for variable initialization and assignment. These `rand()` calls were replaced by a 16-bit Linear Feedback Shift Register (LFSR) that is efficiently implemented in hardware.

In the probSAT C implementation, each variable is initialized by getting a random polarity value using `rand()%2`. Instead of calling the LFSR for each variable assignment, each 16-bit LFSR value is assigned directly to a 16-bit block of the variable space; thus reducing the number of clock cycles needed for variable assignment by a factor of 16. Initial experiments were run with each core being seeded with the corresponding core identifier. It was later realized that seeding them this way could result in the LFSRs becoming synchronized with each other quickly. To remedy this, LFSR seed values for each core were pre-computed to ensure that the $2^{16} - 1$ state space of the LFSR would be even split across the search cores. Theoretically, this would increase the diversity of the initial assignment, and also make it less likely that the solve cores would synchronize to search the same space.

The performance and efficiency following initial transformation was quite poor. Reported latency for each `pickAndFlip` operation could be as long as several thousand cycles. The optimization process mostly consisted of applying Vivado HLS `PIPELINE` and `UNROLL` pragmas and re-structuring the code to reduce data dependencies and reduce initiation intervals. Initially this meant reordering operations and adding registers in the form of `static` variables. Eventually a local cache was created to store the occurrence tables for all candidate variables while calculating the break value. This relieved the performance pressure of accessing the occurrence table while updating the `falseClauses` data structure after flipping a variable allowing for more efficient pipelining of those loops. Some operations that are very efficient on an x86 processor take many clock cycles to complete. For example, the original code uses $flip \bmod numFalse$ as a method to randomly index into the `falseClauses` structure and randomly select a clause to satisfy. Vivado HLS maps this to the llvm `srem` operator which took 30+ clock cycles to execute. This index was replaced by a simple counter that resets to 0 whenever the counter value exceeds `numFalse`, achieving the same effect in a single cycle. The software probSAT implementation also heavily uses `double` floating point representations for probabilities and other values, these were converted to use Xilinx's arbitrary precision fixed-point `ap_fixed` type. Experiments were performed with the C-simulation to decide on the correct fixed point precision. Many integer types were also replaced with `ap_[u]int` types but aside from interface variables this made little impact as Vivado proved incredibly adept at reducing integer types to the minimum necessary bit-widths. Ultimately the maximum latency of `pickAndFlip` on most instances is between 100-250 clock cycles.

Table 3.1: Performance Evaluation: `sgen1` Instances

Instance	MiniSAT	probSAT (SW)	probSATHW(128C)	Speedup (MiniSAT)	Speedup(probSAT)
sgen1-100-100	0.180s	0.168s	0.008s	20.667x	22.144x
sgen1-180-100	519.276s	19.904s	1.044s	497.133x	19.055x
sgen1-200-100	4658.9s	81.4s	13.896s	335.277x	5.858x
sgen1-220-100	1768.36s	380.234s	19.837s	89.147x	19.168x
sgen1-240-100	32444.7s	3912.092s	39.171s	828.277x	99.871x

Table 3.2: Performance Evaluation: `sgen1` Instances with LFSR Spread

Instance	MiniSAT	probSAT (SW)	probSATHW(128C)	Speedup (MiniSAT)	Speedup(probSAT)
sgen1-200-100	4658.9s	81.4s	1.571s	2965.125x	51.806x
sgen1-220-100	1768.36s	380.234s	.006s	290982.929x	62567.352x
sgen1-240-100	32444.7s	3912.092s	395s	82.138x	9.904x

At this point, the memory structures were still treated as external memories and the variable assignment and `falseClauses` initialization procedure as well as the multi-try logic were still completed externally in the C++ testbench. Then the clause databases, occurrence structures, variable assignment storage, and other data structures were moved inside the top-level HLS function as well as simple routines to randomly initialize variables and perform `maxTries` number of tries resulting in a full hardware SAT solver routine. A pre-processor based on the probSAT parsing code populates a header file with the invariable instance attributes and clause database structures. When these structures are accessed from the HLS functions, the appropriate memory structures are automatically instantiated. Finally some optimization was done to reduce resource usage as preventing the unrolling of a short-running loop reduced LUT usage by 3-4x and using the `INLINE` pragma to inline the `pickAndFlip` routine eliminated memory redundancy. Core performance was reduced by 10-20% but these changes made it possible to fit 128 cores on the target device, where only 16 could fit before.

Table 3.3: Average Speedup with 128 Cores

Instance	Speedup (MiniSAT)	Speedup (probSAT-SC14)
`sgen1-[100-240]`	354	33
`sgen1-[200-240]` LFSR Spread	98,010	20,876

Table 3.4: Core Utilization and Performance

Instance	BRAM	DSP	FFs	LUTs	Flips/Sec (HW,1C)	Flips/Sec(SW)	Est. Flips/sec(128C)	Improvement
uf250-1065 (mean)	17	2	1543	2565	1.614×10^6	6.010×10^6	206.657×10^6	34x
sgen1-100-100	5	1	1677	2651	1.861×10^6	14.298×10^6	238.146×10^6	16x
sgen1-200-100	7	1	1719	2643	NA	13.888×10^6	NA	NA
sgen1-220-100	8	1	1722	2652	NA	18.166×10^6	NA	NA
sgen1-240-100	8	1	1690	2535	NA	NA	NA	NA

The overview of a single solver core is shown in Figure 3.1. The details of the `pickAndFlip` core are shown in Figure 3.2, and the many-core solver architecture built from these is shown in Figure 3.3. The architecture as presented is application-specific. Any instance that can fit in the allocated memory blocks can be solved by the presented architecture.

One design decision was for each core to be fully independent with direct access to a dedicated memory store. This required some experimentation to find the largest supportable instance before memory requirements become untenable for BRAM implementation of many cores. With the current memory encoding this solution is limited to around several hundred variables when fitting 128 cores on a single device and testing focused on small but difficult problems.

3.3.1 OpenMP Implementation of probSAT

In order to more rapidly prototype configuration changes and easily test the solver approach against larger problems, an OpenMP/C++ implementation of the probSAT solver was developed.

Algorithm 2 Algorithm for OpenMP parallel solver

```
$cost \leftarrow cost(F, \alpha)$
$curIter \leftarrow 0$
$sat \leftarrow 0$
#pragma omp parallel for
while $curIter < maxIters$ and $sat == 0$ do
    for $curTry = 0$ to $maxThreads$ do
        if $curIter == 0$ then
            $workerArray[curTry] \leftarrow$ new $probSATWorker$
        else
            $\alpha \leftarrow$ random truth assignment
            $score =$ probSatWorker→solve($\alpha$)
            if $score == 0$ then
                SATISFIABLE
                $sat++$
            end if
        end if
    end for
    $curIter++$
end while
```

The main concept is similar to that of the hardware implementation. Each processing core available on the cluster should host a single search thread that has been seeded individually. In order to do this, the primary probSAT functionality was encapsulated in a *probSAT_worker* class. While the central main function still allocates memory, parses the instance and allocates memory for the shared data structures, the probSAT_worker class integrates variable assignment, initialization, and the actual search. An array of worker objects was allocated equal to the number of processing cores P. The initial implementation was very simple with a `for` loop executing P iterations. This loop was parallelized using the `omp parallel for` construct to execute each loop of the solver routine in parallel. The pseudo-code for this approach is shown in Algorithm 2.

The OpenMP implementation replaced the 16-bit LFSR with a 32-bit Galois LFSR, and a detected performance bottleneck due to the non-reentrant nature of the glibc `rand()` function motivated the investigation of alternate PRNGs for stochastic behavior as discussed with more detail in Chapter 4.

3.3.2 Hybrid OpenMP/MPI Implementation of Parallel probSAT

The initial OpenMP implementation of this solver limited the available number of cores to those available on a single node. Additionally, there were performance anomalies when using shared memory for the clause database. In order to scale up to be able to use more cores, a hybrid architecture was developed that would run one MPI task was per node and each MPI task would use the OpenMP construct defined in Algorithm 2 to spawn N search threads where N is the number of cores available on a single CPU socket. In order to ensure local access to shared memory, the `numactl` utility was used to assign socket affinity to each MPI task based on the tutorial in [109]. With each MPI task sharing socket-local memory amongst its OpenMP threads, flips/second per thread remains fairly constant as the number of threads increases.

The hybrid approach differs from the one presented in [8] which used MPI to spawn independent search tasks. The approach presented here offers a more natural abstraction of the underlying hardware architecture, OpenMP is used where threads are sharing locally attached memory (each socket), and MPI is used to spawn independent tasks when cores are seperated by a Quick Path Interface (QPI) or Infiniband link. The main benefit of the hybrid approach is in terms of memory efficiency. The clause and occurence databases are constant and are the largest in-memory structures in the *probSAT* implementation; sharing these structures among local threads can reduce overall memory consumption by a factor of 12 or 16 depending on the underlying compute resources.

3.4 Results

An Alpha-Data ADM-PCIE-7V3 card with a Xilinx Virtex-7 XC7VX690T device was the target platform for the FPGA implementation. Desired operating frequency of the probSAT core was 200MHz and after the optimizations mentioned in Section 3.3 the estimated HLS clock period was 4.89ns. It was possible to implement 128 cores while achieving timing closure at 200MHz; up to 200 cores could fit onto the device but would not meeting timing with the default Vivado implementation strategies. Area utilization of the HLS core applied to various benchmark instances is shown in Table 3.4. These utilization estimates are based on the output of Vivado HLS for a single core.

In order to evaluate the performance of this solver benchmark instances from the `uf250-1065` suite[67] and `sgen1-sat` suite[126] were used. The `uf250-1065` suite of benchmarks are relatively easy problems that are solved quickly, even in hardware co-simulation. Co-simulations were run against the entire suite and results were averaged as shown in Table 3.4. The primary metric for these benchmarks is expressed in terms of flips/second as actual run-time for all three solvers tested (MiniSAT, probSAT, and probSAT HW) is near zero. On the other hand, the `sgen1` problems are incredibly difficult for their size. Instances as small as 240 variables take MiniSAT 9+ hours to solve. The long run-time makes it impossible to simulate these in a reasonable amount of time. Instead these were implemented and tested in hardware with 128 cores with results as shown in 3.1. These results show a max speedup of 99.8x over probSAT (Sat Competition 2014 version) and 828x speedup over MiniSAT (v2.2 with DAG simplification) running on an Intel Core i5-4670K processor with 32GB of DDR3 memory. Average speedup on the `sgen1` instances is 32.9X and 354.4X compared to probSAT and MiniSAT respectively as shown in Table 3.3. Due to the non-determinism of the software probSAT implementation, the runtimes shown are an average of five runs for each instances. When discussing the results, speedup is considered super-linear if the speedup factor exceeds the expected increase in overall flips/second.

The modifications to deliberately spread the LFSR state space across the threads exhibit super-linear speedup for the `sgen-200` and `sgen-220` instances, with a maximum speedup of 62567x over the sequential probSAT algorithm on the `sgen-220` instance. On the other hand, solving of the `sgen240` problem slowed down by an order of magnitude compared to the original seeding procedure. Both of these results seem to indicate some element of luck in terms of the initial assignment being close to the solution. The average speedup over MiniSAT and probSAT is 98,010 and 20,876 respectively for the LFSR spread variant as shown in Table 3.3.

The speedup over probSAT would appear to be super-linear compared to the increase in flips/second. The speedup numbers numbers for the `sgen1-100-10` instance from Table 3.1 compared to the flips/second increase in Table 3.4 show this explicitly. It is possible that there is an advantage in having more cores exploring different parts of the Hamming space. The stochastic nature of the algorithm makes it difficult to confirm the modifications made to the implementation remain true to the probSAT algorithm, however the single-core `uf250-1065` simulations show the solver converging to a verified solution in approximately the same number of total flips with around 9% less flips required for the hardware solver.

Table 3.5: VMPC Benchmark Instances - OMP/MPI Hybrid ProbSAT

Instance	MiniSAT	probSAT-SC14	24T	48T	96T
grieu-vmpc-31	TO	TO	4,744	3,051	**1459**
vmpc_29	62	4639	205	18	**18**
vmpc_33	TO	TO	3457	3410	**421**
vmpc_32.renamed-as.sat05-1919	1438	TO	2989	513	**496**

Table 3.6: PAR10 Speedup with 96 Threads - OMP/MPI Hybrid ProbSAT

Instance	Speedup (MiniSAT)	Speedup(probSAT-SC14)
grieu-vmpc-31	34.3	34.3
vmpc_29	3.4	257.7
vmpc_33	118.8	118.8
vmpc_32.renamed-as.sat05-1919	2.9	100.8

Another class of instances that benefits from the parallel local search solver presented here are the VMPC instances which encode the inversion of the VMPC one-way function presented in [138]. These instances were solved on the Virginia Tech NewRiver compute cluster where each node is equipped with two Intel Xeon E5-2680v3 for a total of 24 threads per node. Each CPU runs at 2.5GHz, but is capable of boosting clock speed up to 3.3GHz on single-threaded workloads. These are relatively hard instances, with the single threaded MiniSAT unable to solve two, and the single-threaded probSAT from the 2014 SAT competition unable to solve three out of the four tested instances within 5000 seconds. On the other hand, the multi-threaded hybrid OpenMP/MPI hybrid solver was able to solve all four instances, with time to solution scaling approximately linearly as the number of threads increased. In all cases, the 96T solver found the solution in the shortest amount of time as shown in bold in Table 3.5. Speedup over MiniSAT and probSAT-SC14 with a 10X penalty for a timeout (PAR10) is shown in Table 3.6. For the `vmpc_29` instance which was the only one solved by both MiniSAT and probSAT-SC14, the 96 thread solver shows a 3.4X speedup over MiniSAT and a super-linear 258X speedup over probSAT-SC14. For these tests each thread was seeded by a 32-Bit LFSR where the state space of the LFSR was split into 128 even segments, and each thread was assigned a seed value that indexed into one of these segments.

3.5 Summary

This work presented a hardware architecture for an effective probabilistic SAT solver based on the probSAT algorithm proposed in [15]. No other hardware implementation of this algorithm could be found in the literature. Kanazawa et. al[71, 72, 73] presented efficient hardware implementations of WalkSAT that exploited fine-grained parallelism through parallel clause processing, but they did not explore massively distributed local search with independent search threads. On the other hand [8] explored the use of massive SLS portfolios up to 512 threads, but their testing focused entirely on general purpose compute resources. The MPI-based approach used in [8] would also result in redundant memory structures compared to the hybrid OpenMP/MPI solver presented her in Section 3.3.2. The primary contribution presented here is a massively parallel SLS SAT solver with a corresponding efficient hardware implementation.

The hardware implementation takes advantage of fine-grained parallelism through the use of clause and occurence processing pipelines. A compact search core was developed using HLS to rapidly port the C implementation of the algorithm provided at [13]. A basic port was functional within the first full week of working with the code and the implementation was optimized, tested, and refined over several months. The core was simulated over a variety of small but relatively difficult SAT instances to check for correctness and performance. A 128-core prototype of the design was implemented and tested in hardware showing strong performance compared to MiniSAT and the software implementation of probSAT on the tested instances. Further tests with the cluster implementation showed strong performance on a set of hard instances that encode the inversion of the VMPC one-way function, but performance was worse than sequential CDCL algorithms on most other industrial application instances. The `sgen1` and `vmpc` instances where the parallel local search solver performed well were both examples of instances classes that are difficult to solve despite being relatively small; the largest of these instances (`vmpc-33`) contains only 1089 variables.

Future work on this approach would include capacity expansion of the hardware solver. Capacity could be increased by using more efficient memory encoding techniques for the clause database as shown in [128]. Alternatively, the design could be modified to work on blocks of clauses at a time with support for multi-threading to hide the latency of high throughput off-chip memory such as a Hybrid Memory Cube (HMC) or a High-Bandwidth Memory (HBM) stack as the central clause store. The implementation tested here also presented a 128-core design that was re-implemented for each instance. A true application specific implementation would generate a core that could handle the largest possible instance in an application class and generate a fixed structure with the maximum number of cores that could be loaded by programming new memory contents. The compute density and performance of the hardware implementation would also be greatly increased by using a newer FPGA such as the Xilinx XCVU9P devices that boast greater logic capacity, higher clock speeds, and UltraRAM resources to store larger instances without having to resort to high-latency off-chip memory. The OpenMP/MPI software variant can be used to more flexibly profile classes of instances and tune algorithms and parameters for specific instance classes. The implementations presented here used the default configuration parameters from the probSAT source available on Github[13] that were tuned using random instances, it is possible that specific tuning for application instances will increase performance beyond the small, hard satisfiable instances presented here.

Chapter 4

Efficient Seeding of SLS Solvers to Distribute Search Space

Recent developments in Stochastic Local Search (SLS) solvers for Boolean satisfiability have resulted in extremely effective solvers for random, crafted, and even hard-combinatorial SAT instances as shown in the last several SAT competitions[4]. Specifically probability distribution SLS solvers such as probSAT[15], Sparrow[14], and YalSAT[26] have shown strong performance in recent SAT competitions. Part of the success of these solvers is the focus on efficient implementation and empirical tuning of simple probabilistic algorithms, a thorough discussion of the implementation details is presented in [12]. It is possible to trivially parallelize SLS for SAT by distributing independent solver threads across solver cores. A typical approach to this would be to seed each solver thread with a "random" initial assignment, ideally allowing each thread to traverse a distant portion of the vast search space. The pprobSAT solver won the parallel random SAT track in 2014 using this approach. A similar approach was shown in [8] to be scalable up to 512 threads, at least on crafted and verification (`cbmc`) instances. Interestingly, adding an information sharing mechanism, even an efficient one with low overhead seemed to hinder scalability past 16 cores in [7].

The lack of communication overhead makes SLS solvers an interesting choice for scaling concurrency for SAT on compute clusters or even Field Programmable Gate Arrays (FPGAs) as shown in [124] and discussed further in Chapter 3. One aspect of SLS SAT implementations that hasn't been explored in recent publications is the choice of Pseudo-Random Number Generators (PRNG) used for initial assignment; most implementations simply use the default PRNG supplied by their platform, typically the **rand()** function provided by **libc**.

In [124] a simple 16-bit Linear Feedback Shift Register (LFSR) was used as a source for stochastic behavior and showed very strong performance with good scalability on the small but difficult **sgen1** class of crafted benchmarks. An attempt to model [124] with a multi-threaded OpenMP implementation of probSAT exposed the non-reentrant **rand()** function from **glibc** as a major bottleneck with up to $\frac{2}{3}$ of CPU time spent synchronizing access to PRNG state. Using the reentrant **rand_r()** function resolved the bottleneck, but the consequent reduction in state space and relatively poor statistical soundness of **rand_r()** motivated an investigation into the significance of PRNG selection for multi-threaded SLS solvers.

This chapter considers the performance tradeoffs both in runtime and quality of results between the simple and very fast Linear Feedback Shift Register (LFSR), a slightly more complex but slower Linear Congruential Generator (LCG) provided by **glibc rand_r()**, and the statistically proven yet performant Permutation Congruential Generator (PCG) family of generators described in [99] and available at [101]. A C++ implementation of probSAT utilizing OpenMP to spawn independent worker threads is used as the test-bench.

This chapter further presents how considered and deliberate seeding of the SLS solver threads can bring benefits such as an increased range of Hamming weights across assignments, efficient hardware implementations, and potentially even completeness after a number of restarts without requiring communication between threads. Several different methods for deterministic, distributed seeding procedures, with multi-bit assignments are presented and their output and performance is compared to more traditional techniques that use standard PRNGs to assign each bit individually. Performance is compared both with real-world benchmark instances and simulations measuring minimum, maximum, and average Hamming distances and Hamming weights across assignments.

4.1 Deterministic Seeding

Traditionally SLS solvers initialize variable assignments with a random or pseudo-random value. Most algorithms incorporate a cutoff points, or a maximum number of flips before re-initializing with another random assignment, this is referred to as a *restart* of the search. This is a reasonable approach as in a 2^N search space, it is improbable that any two variable assignments will be near each other in the search space.

The hardware solver presented in [124] used a 16-bit LFSR as a PRNG and in the interest of reducing execution time and hardware utilization, variables were assigned the LFSR value in 16-bit blocks instead of assigning them one-bit at a time. One limitation of the approach used in [124] is the possibility for LFSR states to overlap, effectively synchronizing the solver cores at the next restart. One approach to avoid that scenario is to divide the LFSR state space across the cores by pre-calculating the start state for the i^{th} core state as the $(2^{16}/N)*i$ state of the 16-bit LFSR from a fixed,common seed. Eventually the seeding procedure was updated to do so and as discussed in Section 3.4, this had great performance benefits on a couple of the tested instances.

4.1.1 Completeness Through Seeding

A Boolean formula with N variables contains a search space of $2^N - 1$ solution candidates. Of course, exploring such a search space naively without some sort of heuristic is infeasible. In this section, there will be a discussion of some ways to have a provably complete, deterministic SLS solver using clever metrics to distribute the Hamming space over the the available parallel cores.

In [66] a proof was made based on Hamming distance that proves the concept of probabilistic approximate completeness. Essentially for any assignment a, there exists a flip decision that would reduce the Hamming distance h between a and the solution s by at least one. Using that lemma inductively, they prove that there exists a sequence of h random walk steps that would lead to the solution s from a, and they are able to bound the length of this random walk proving that as the numbers of steps $t \to \infty$, the probability of reaching the solution approaches one. Other works such as [116] are able to calculate provable bounds for random-walk local search algorithms on specific instance classes such as 3-SAT based on a similar concept of Hamming distance and Markov chains. In [37], a deterministic local search algorithm for k-SAT was proposed with a complexity of $poly(n) * 1.481^n$ for 3-SAT. This algorithm eschews random assignments for a covering code that splits the search space into Hamming balls and conducts local search within those Hamming balls of reachable assignments within the given bound. The approach from [37] was further improved upon in [28] with a lower bound. Note that the algorithms in [28, 37, 116] will find solutions if they exist, they are not complete in the sense that they are unable to prove unsatisfiability.

The deterministic set covering approach taken in [28] and [37] is particularly interesting because the covering code is calculated in polynomial time and the subproblems generated for local search can be carried out completely independently, potentially concurrently. While such an approach would seem to have potential in terms of parallel acceleration, at least for random 3-SAT, it is outside the scope of this investigation. Instead we focus on approximating this affect with existing local search algorithms by attempting to spread the Hamming space across cores.

Complete Search with N-Bit LFSR

Consider a PRNG with a state space of $2^N - 1$ and output width of N bits where it is possible to precompute, or compute in $O(1)$ the i^{th} state. With a linear PRNG such as an LFSR, it is possible to split the search space across cores without sharing any data. Instead the starting state of each core and potentially the number of LFSR states are skipped at each restart are determined by the core identifier number. If, in aggregate the processing cores then exhaust all possible states of the LFSR, we have a provably complete local search algorithm. More specifically a pattern generator that guarantees a minimum Hamming distance between states could be used to ensure a good spread in distance between states. Such concepts have been proposed before in the Automatic Test Pattern Generation field[107] and are relatively efficient in terms of hardware consumption.

Attempted Seeding Mechanisms

The above approach of an N-bit LFSR with Hamming distance guarantees between states may be inefficient to implement in software on 64-bit CPU architectures. Several different approaches were tested here to approximate the effect in software, albeit without the completeness guarantees.

One approach that was considered is to use a K-dimensional PRNG such as the ones provided within the PCG family of PRNGs[99]. The PCG suite offers arbitrary K-dimensional, equidistributed PRNGs with a guarantee that over the period of the PRNG every possible K-tuple of 32-bit words will occur the same number of times. One seeding strategy tested here uses a 64-dimension PCG PRNG to initialize the assignment for each thread. At each restart the thread is seeded by advancing the PRNG by $(iteration + thread\_id) * 64$ states, ensuring that each thread gets a unique seed value for the current local search phase. Additionally, the equidistribution guarantee over the 64 dimensions ensures good statistical distribution of assignment values for problems with approximately 2048 (32x64) variables. The equidistribution guarantee in 64 dimensions ensure that within the 2^{2112} period of this generator, every possible 2048 bit value will be generated. For problems with less than 2048 variables, a full cycle of the PCG state space would also guarantee completeness albeit in an intractable amount of time. The PCG family of generators also support creating generators with arbitrary dimensionality to better tailor number generation to the number of variables, but this cannot be done dynamically. Perhaps the **oneseq** variants of these generators would have been more appropriate here in terms of guaranteeing an exhaustive search, but they were not tested here. Additional work could have be done to store a bank of the N-dimensional generators in order to scale across cores.

Another approach extends the distributed state-space 16-bit LFSR discussed in Chapter 3 to 32 bits. Each worker thread contains it's own 32-bit LFSR seeded from pre-computed values intended to split the $2^{32} - 1$ state space into 128 even portions ensuring a unique initial assignment of variables for problems with up to 2^{25} variables. Additionally, the linear relationship between LFSR states generally enforces a large Hamming distance jump between assignments on different cores, even upon restart.

Finally, an approach referred to as the *spreadHam* method was developed for this work. This approach attempts to maximize the range of Hamming weights of initial assignments across all search cores while still guaranteeing a minimum separation of Hamming distance between any two cores. This is done very simply by initializing each thread with a unique identifier that is $m = log2(num\_threads)$ bits wide. This value is then applied to each m-bit block of the variable assignment. What this effectively does is ensure that each thread starts at least N/m bits away from each other in Hamming distance, while also covering the extreme corners of the Hamming space such as all-zero and all-one assignments which may have special significance in encoding real world problems. This method was applied to the WPMS solver presented in Chapter 6 and shows good scalability in terms of solution diversity up to 128 threads. The spreadHam assignment is only used for the initial assignment, and one of the other methods is used on subsequent restarts.

A major consideration that makes all of these approaches suitable for scalable, parallel implementations is that each thread only needs to know its own ID, no communication is required upon restart to ensure a unique assignment.

This concept of planned seeding of SLS cores differs from solvers such as **probSAT** where the initial assignment of each variable generated by calling **rand()%2**. It is proposed here that using deterministic methods as described above provides potential benefits to massively parallel SAT solvers by efficiently distributing the Hamming space across search threads, and ensuring that each thread starts and restarts in a different portion of the search-space. Instead of focusing on uniform distribution of assignments across cores, seeding methods such as the *spreadHam* strategy are deliberately non-uniform in order to ensure that distant corners of the Hamming space such as all-zero or all-one assignments are explored on each run. Additionally some seeding methods such as an N-bit LFSR offer the potential for complete, deterministic local search when combined with an aggressive restart strategy.

4.2 Methodology

The investigation of different PRNGs and seeding methodologies started with an attempt to model the approach presented in [124] on a general purpose computing cluster using OpenMP. While there is a parallel implementation of probSAT, **pprobSAT** that was submitted to the 2014 SAT Competition, the approach of spinning off threads using Python's **subprocess** module did not appear to correctly spawn threads across nodes in a cluster environment. Instead, a parallel OpenMP port of probSAT was created using C++. The core probSAT solver code and data structures were encapsulated in a **probSAT_worker** class. The constructor for this class copies the initialized clause database, and occurence table structures from the single-threaded parser. Each class instance maintains private copies of solver state and assignment-dependent data structures. The OpenMP **#pragma parallel for** construct is used to spawn independent threads that construct their own **probSAT_worker** objects with differing initial assignment values. This implementation is discussed in more detail in Section 3.3.1. The worker threads each use the same tuned parameters from the version of **probSAT** hosted on GitHub [13]. One significant difference is that instead of using tuned values for **maxFlips** and **maxTries** here we simply set **maxFlips** to $2E9$ and keep restarting until the timeout is reached. As opposed to the 5000s timeout used in the SAT competitions, here a 300s timeout is used.

4.2.1 Replacing rand() in Flip Decision

For **probSAT** and related algorithms each flip decision requires the generation of decimal value between 0 and 1 that can be compared to the probability masses that are pre-calculated from the chosen Probability Density Function (PDF).

It was observed during the implementation of the parallel OpenMP variant of probSAT presented in Section 3.3.1 that the **glibc rand()** function is not thread-safe and created a significant performance bottleneck as **glibc** locks kept threads waiting for a random value up to $\frac{2}{3}$ of the multi-threaded runtime. By default, the Linux implementation of **rand()**[48] uses a relatively sophisticated PRNG algorithm[117] with 128-bytes of state, but the reentrant **rand_r()** variant only supports 32-bits of state with a simple Linear Congruential Generator which may be less statistically sound. It wasn't clear that statistical soundness is a concern for this application, however, as the simple 16-bit LFSR used in [124] showed strong performance on crafted instances. In order to clarify the significance of statistical soundness in the flip mechanism of SLS solvers, experiments were performed to compare different sources of flip randomness including modern PRNGs such as [101] with strong claims of superior statistical soundness.

A set of benchmarks compared the performance of the **rand_r()** LCG with 16-bit and 32-bit Galois LFSR functions as well as 32-bit variants of the PCG family. Different executables were compiled using each PRNG. Initial attempts to use a function pointer to dynamically change the PRNG adversely affected performance as it prevented the function calls from being inlined. A Python script iterated over all the CNF files in a given folder and submitted a cluster job for each executable. The benchmarks were run on the Virginia Tech NewRiver cluster where each node contained two Xeon E5-2680v3 CPUs for a total of 24 cores. The solver variants were all compiled with the same aggressive optimizations used by **probSAT** in the 2014 SAT Competition.

4.2.2 Hamming Distance Between Threads and Hamming Weight Range Across Threads

Traditionally, parallel SLS solvers aim to uniformly distribute assignments across cores. This is typically done by using the least significant bit of a PRNG output to affect a binomial distribution with $p = 0.5$ assuming the PRNG is statistically sound.

While such an approach should ensure good diversity in terms of Hamming distance between assignments, it is extremely unlikely to generate assignments near extreme corners of the Hamming space, including assignments with Hamming weights of 0 or N. Conversely, a non-uniform deterministic method such the the *spreadHam* approach can cover distant corners of the Hamming space while still offering guarantees in terms of Hamming distance between assignments.

In order to compare how different seeding methods might affect Hamming distance between assignments and the overall range of Hamming weights covered, a set of simulations were performed comparing methods that assign one bit at a time with **rand()** and **pcg32** as random number generators to multi-bit deterministic methods including the LFSR32, PCG32K64, and spreadHam methods described in Section 4.1. The simulation is implemented as a C++ application that takes the seed method and number of variables as arguments and generates 128 assignments. The min, mean, and max values for Hamming distance between assignments, and Hamming weight across assignments were recorded. Results for the non-deterministic, randomly seeded **rand** and **pcg32** methods were averaged over 10 runs. Both methods use the **std::random_device** construct to generate the initial seed. For the **rand()** function, the least significant bit is extracted with the C++ **%** operator, whereas the PCG PRNG has in built support for a binary output range by passing an argument to the function, i.e. **my_pcg32(2)**.

4.2.3 Comparison of Seeding Methods

In Section 4.1 there are multiple methods discussed for deterministic seeding of the initial assignment for distributed local search. The methods that were implemented as part of the OpenMP/MPI implementation of the solver presented in Section 3.3.2 include the *LFSR32*, *PCG32_K64* and *spreadHam* methods. The LFSR32 method precalculates an array of 128 seeds that split the 2^{32} state space of the 32-bit LFSR into 128 even portions, and each 32-bit value is applied to 32 variables at a time. The LFSR for each thread is the $(2^{32}/128) * tid$ state of the LFSR where tid is a unique thread identifier from zero to 127. The PCG32_K64 seeding method utilizes the 64 dimension random number generator from the PCG family of PRNGs. This generator guarantees equidistribution over all possible 64-word sequences, and thus each thread advances the PRN by $64 * (iteration * tid)$ which allocates each thread a unique 64 word sequence of 32-bit words that are evenly distributed across a 2048-bit Hamming space. The spreadHam method splits the assignment space into chunks of $X = log_2(M)$ bits where M is the number of allocated cores. Each core assigns its thread ID to each X bit chunk which ensures that the minimum Hamming distance between any thread assignment is at least $N/log_2(M)$ bits where N is the number of variables.

The pprobSAT solver as submitted to the 2014 SAT competition took a different seeding approach. This solver used a master python script that spawned independent instances of the probSAT solver. The python **rand** mechanism is used to generate M seed values that are passed to each probSAT instance. Each probSAT instance then uses this value to seed GNU **rand()** which is used to assign each bit individually using $rand()\%2$ to get a binary assignment. This approach is approximated here by seeding GNU **rand()** with **srand(time(0))**, and using **rand()** to generate M seed values for each of M threads, these seed values are then used to seed the reentrant **rand_r()** function. The least significant bit of the **rand_r()** output is then used to assign each variable.

Performance of the various seeding methods were compared using the `vmpc-33` instance with 96-threads. The LFSR32 and spreadHam methods are fully deterministic and will give the same results each time. For the **rand_r()** based seeding methods, the solver was run 10 times and the min, mean and max time to solution are calculated.

4.3 Results

4.3.1 Evaluation of PRNG for flip decisions

PRNG Variant	Flips/Second
LFSR16	66459376
LFSR32	65338688
RANDR	63689001
PCG	65892941
PCG-FAST	64796260
PCG-ONESEQ	65990734

Table 4.1: Performance in terms of Flips/Second

The two sets of benchmarks chosen to test PRNG performance are the crafted `sgen1` instances as well as a selection of 60 Random 3-SAT (`unif-k3`) instances used in the Random track of the 2014 SAT competition. The timeout for these experiments was set to 300 seconds instead of the 5000 second timeout used for SAT competitions. Table 4.1 compares the performance in terms of composite flips/second over all threads. Table 4.2 shows the number of solved instances as well as the 10X Penalized Average Runtime (PAR10).

Table 4.1 shows that there isn't a large gap in terms of flips/second with the fastest variant (LFSR16) being around 4% faster than the slowest variant (RANDR). It is interesting to note that the PCG-FAST variant is slower than PCG despite the developer's intention that the **pcg32_fast** generator would be a faster but less statistically sound version of the **pcg32** generator.

Benchmark set	LFSR16	LFSR32	RANDR	PCG	PCG-FAST	PCG-ONESEQ
sgen1-(60-240) AR	3.07	2.85	5.27	3.31	239.4	6.55
sgen1-(60-300) (S/13)/PAR10	10/705	10/713	10/701	10/699	9/932	10/699
unif-k3 (60 instances)	22/1931	23/1884	24/1837	20/2025	22/1930	21/ 1980

Table 4.2: PRNG Comparison with Parallel probSAT

There were 13 `sgen1` benchmark instances, and 10 of them (`sgen-60` through `sgen-240`) were solved by all tested solvers aside from the the variant using the PCG-FAST generator. The corresponding results were split up in Table 4.2 with the first line allowing a comparison average runtime across commonly solved instances, and the second line shows overall performance on the set of 13 tested instances.

On the `unif-k3` instances selected from the 2014 SAT Competition, the solver using **rand_r()** solved the most instances (24) followed by the LFSR32 variant (23). The **rand_r()** solver also won in terms of par10 runtime, but was slower than the LFSR32 solver on commonly solved instances.

In terms of the **pcg32** variants, the standard variant is supposed to be more statistically sound while the "fast" and "oneseq" variants are intended to be faster with some tradeoffs in terms of statistical quality. In the tests above however, the standard **pcg32** variant was between **pcg32_fast** and **pcg32_oneseq** in terms of flips/sec, and exhibited inferior performance on `unif-k3` instances and superior performance on `sgen` instances.

Overall, the statistical quality of the random number generator used does not seem to have a significant impact performance. The more statistically sound PCG generator was outperformed by the simpler LFSR32 and RANDR variants, even without a large difference in flips/second.

Table 4.3: Comparison of Seeding Methods on `vmpc-33` Instance with 96 Threads

Seeding Method	Min	Median (PAR10)	Max	Mean (PAR10)
RANDR%2 (10 runs)	1724	3762	TO	17181
PCG32(2) (10 runs)	107	585	1543	709
LFSR32	421			
spreadHam	857			
PCG32_K64	104			

4.3.2 Evaluation of Seeding Methods

A comparison of the various seeding methods used for solving the `vmpc-33` Instance using 96 threads is shown in Table 4.3. The method using the re-entrant **rand_r** function shows inferior performance as the minimum solve time over 10 runs was slower than the maximum solve time for any other method. A similar method using the PCG32 PRNG shows competitive performance with a minimum runtime matching the best deterministic seeding method, and a mean runtime better than the worst deterministic method.

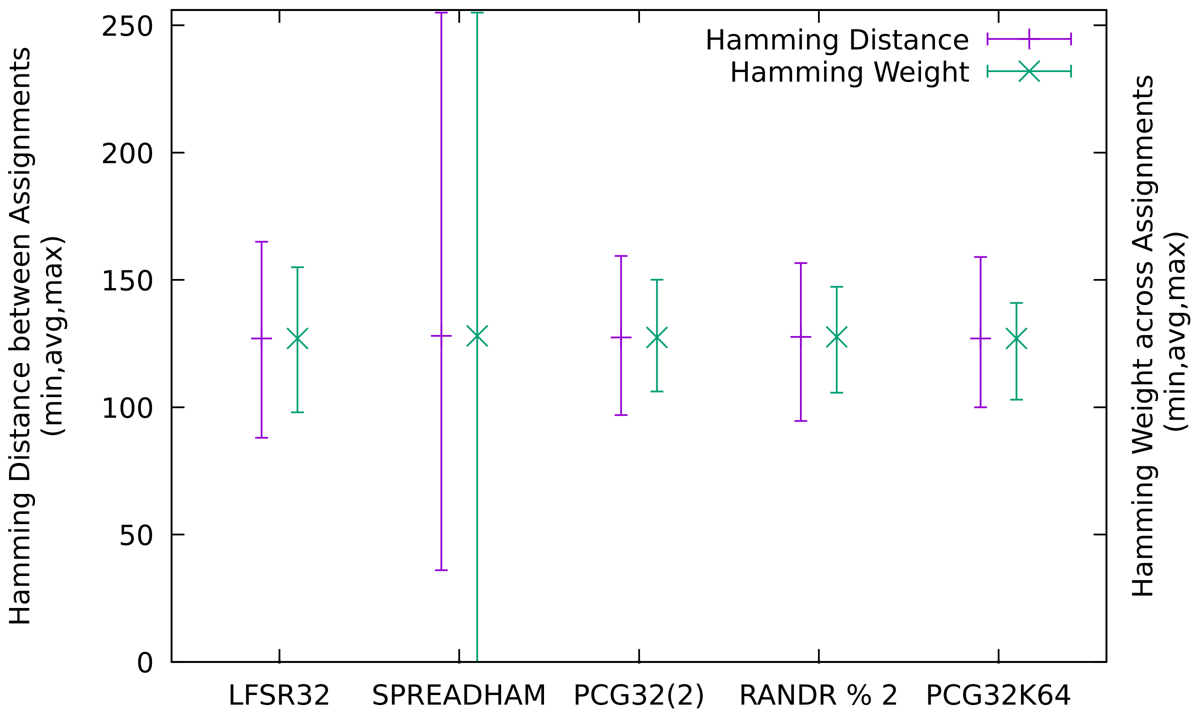
The results of simulations performed to compare assignment diversity among seeding methods are shown in Figure 4.2. The graphs illustrate the minimum, maximum, and average Hamming distance between each of the 128 assignments as well as the overall range and average Hamming weight across the assignments. Problem sizes sizes of 256,512,1024, and 2048 variables were simulated. As expected, the average distance and weight for all methods was around $N/2$, however the spreadHam method in particular with its non-uniform distribution had significantly lower minimum and higher maximum Hamming distances and weights than the other methods. In fact the range of the spreadHam method for all simulations was from zero to N, but the trade-off is a much lower minimum Hamming distance between any two assignments. The LFSR32 seeding method offered a slightly larger range of Hamming weights with much less of a penalty in terms of minimum Hamming distance between assignments. These results show that it is possible to use deterministic, correlated methods to distribute assignments across the most distant corners of the Hamming space while still achieving an average Hamming distance of $N/2$ between assignments and average Hamming weight of $N/2$ across assignments.

4.4 Summary

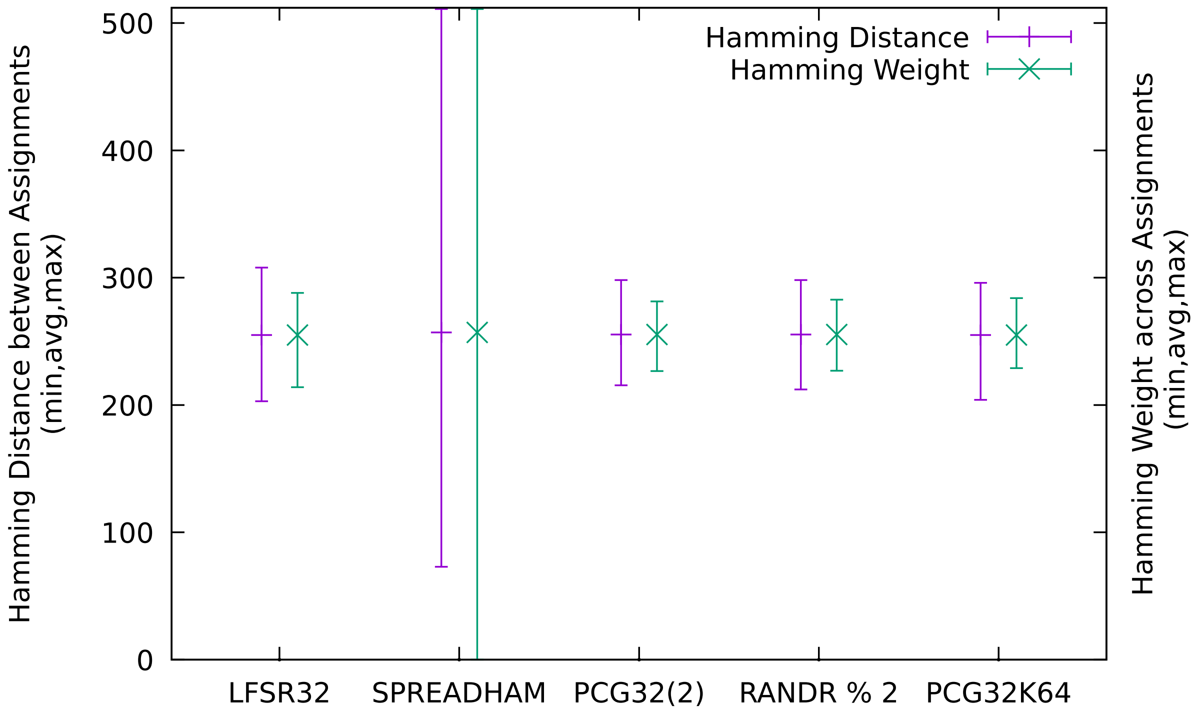
The unsuitability of **glibc rand()** for multi-threaded SLS solvers developed with OpenMP, combined with the strong performance exhibited by the simple LFSR16 implementation in Chapter 3 motivated an investigation into the importance of PRNG selection when used as the source of stochastic behavior in the probSAT algorithm. The results show that the simple one-line Galois 16-bit and 32-bit LFSR offers competitive performance to more advanced PRNGs on difficult random 3-SAT instances from the 2014 SAT competition. Additionally, the difference in rankings between the `sgen1` instances and the `unif-k3` instances show that there may be room for tuning the solver for specific applications by changing out the source of stochastic behavior. This evaluation of the different PRNG methods is a contribution that should help inform developers of SLS search algorithms for SAT.

Deterministic methods for seeding initial assignments across threads where presented in Section 4.1. Significantly, these methods do not require communication between threads. These methods avoid excessive calls to the PRNG by applying the $N-bit$ PRNG output to N bits at a time. The presented methods include LFSR16, LFSR32, PCG32K64 as well as the spreadHam method. The spreadHam method in particular generates assignments spanning the entire range of Hamming weights from zero to N while maintaining guarantees about minimum and average Hamming distance between assignments. Simulation results were presented to illustrate the differences in seed diversity across different problem sizes. This

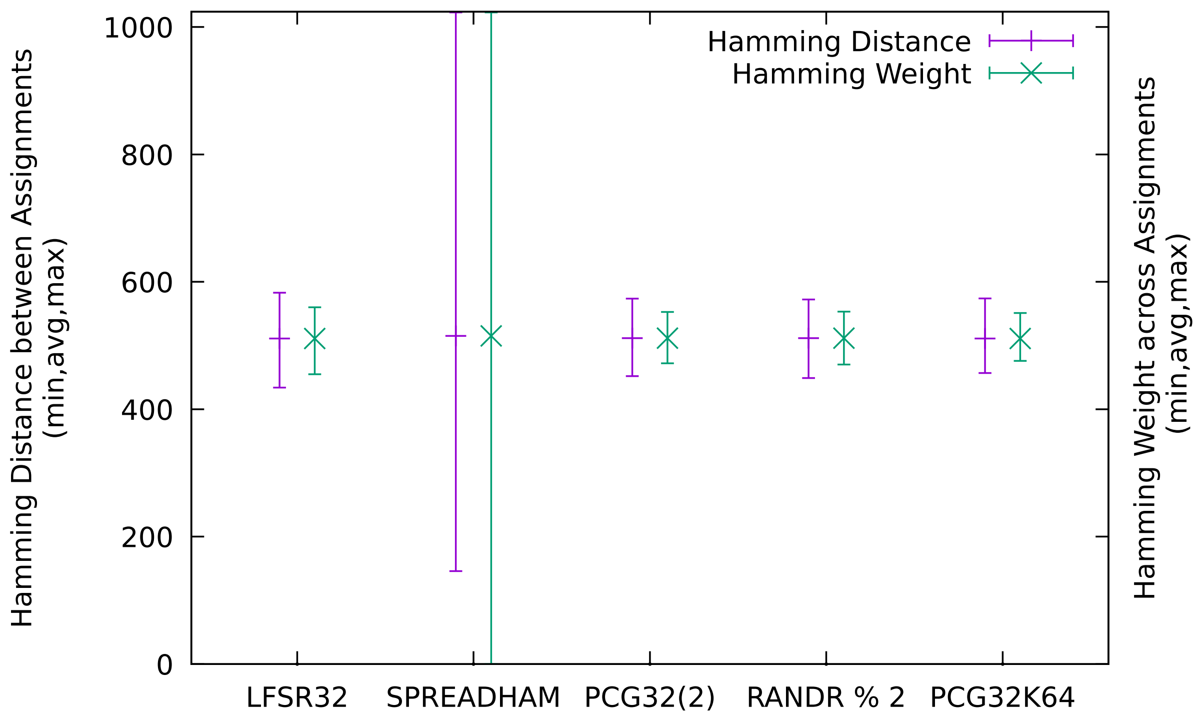
Figure 4.1: Comparison of Assignment Diversity Amongst Seeding Methods



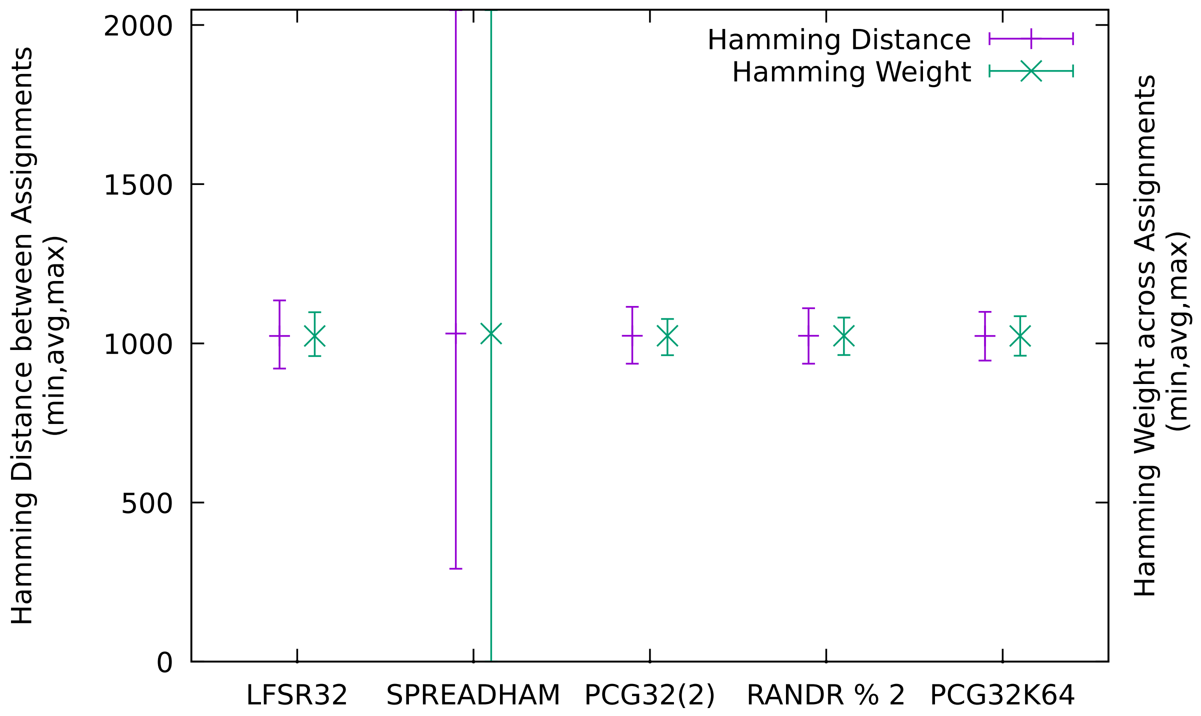
(a) 256 Variables



(b) 512 Variables



(c) 1024 Variables



(d) 2048 Variables

contribution of low-overhead, efficient, and deterministic seeding methods with Hamming distance guarantees will hopefully advance the prospects of massively parallel SLS SAT.

Chapter 5

Concurrent Completeness without Communication Overhead: A Hybrid Approach to Distributed Local Search

There have been many attempts to create a hybrid solver that combines the strengths of SLS and CDCL solvers with some like [50] showing promising speedups on application instances. In more recent SAT competitions, hybrid solvers that don't share information between the SLS and CDCL portions have been successful in the hard-combinatorial tracks[4]. Hybrid MaxSAT solvers such as maxroster[127], the dominant incomplete solver from the 2017 Competition, tend to use local search to determine a reasonable upper bound and then use that upper bound as a starting point for the complete, linear search based solver.

While the distributed SLS search based solver presented in Chapter 3 shows good performance on some instance classes, it is ultimately limited by the fact that it is incomplete, and that SLS solvers in general are not competitive in solving many real-world application instances.

5.1 Attempted Approaches

Several different approaches were tested in pursuit of a solver that would outperform both SLS and CDCL algorithms on some instance classes. The common thread of all these approaches was the combination of the OpenMP SLS solver based on probSAT[15] that was presented in Algorithm 2 with prominent CDCL solvers that implement the **ipasir** interface available from [17].

The initial approach was effectively a multi-threaded version of the iterative solver presented in [50]. The main difference aside from the use of newer SLS (probSAT instead of WalkSAT) and CDCL (Riss instead of zChaff/MiniSAT) solvers is that at each iteration, all SLS threads contributed their unsatisfied clauses to a single pool of clauses to be solved by the CDCL solver. At the end of each SLS iteration, the set of unsatisfied clauses from SLS local minima are passed to an incremental CDCL solver. The CDCL solver finds a solution to this sub-problem (or declares it UNSAT) and the partial solution is used to re-seed the local search threads. Initially all search threads were restarted from the partial solution, but this resulted in all SLS threads eventually being seeded identically and searching the same space. Eventually a single thread used the partial solution as initial seed and the rest of the threads used a random seed as before. On hard instances such as the `aes_24_4_keyfind_5` instances that went unsolved by many solvers in recent SAT competitions, it was clear that this approach would "converge" upon a solution as it rules out various local minima, and eventually the CDCL solver gets stuck at a point where over 90% of clauses were added to the sub-problem and the SLS solver only had one unsatisfied clause. An example of this convergence is shown in Figure 5.1. The largest downside of this approach was that the SLS threads sat idle while waiting for the CDCL solver to complete.

In order to ensure that all available threads remained active while solving, the hybrid solver

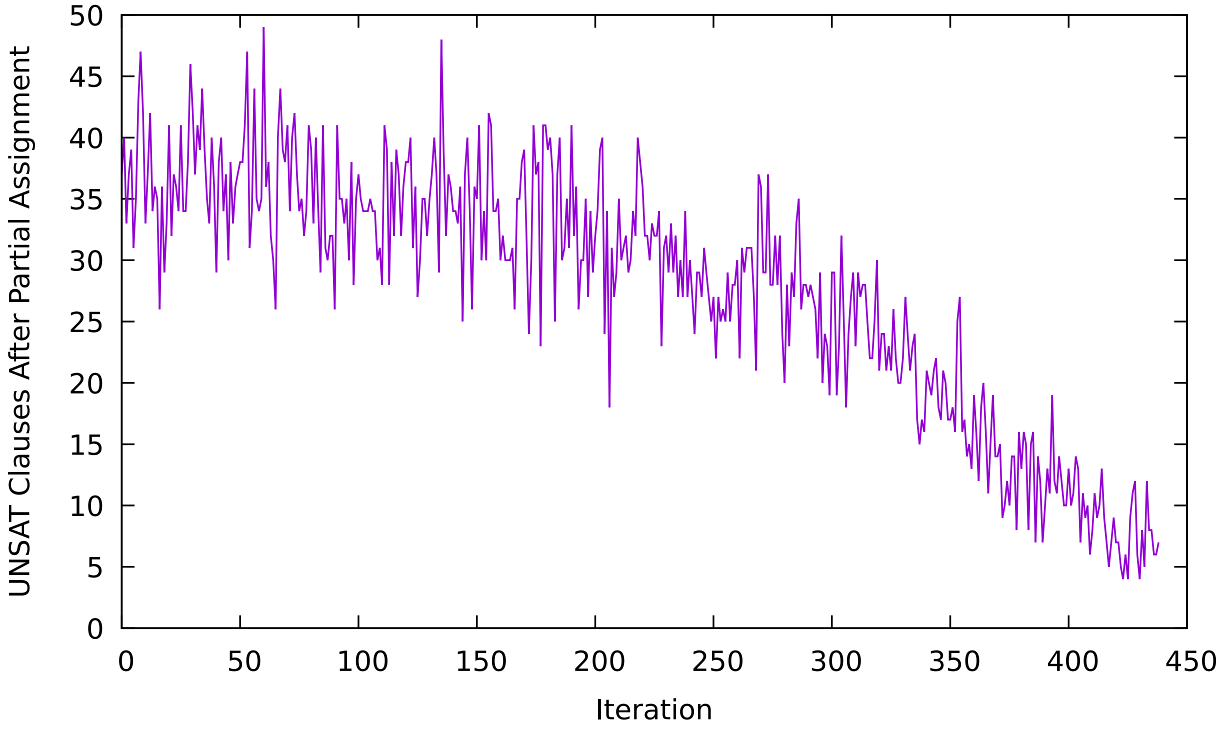


Figure 5.1: Convergence of Many-Core Hybrid Solver on AES24 Instance

was redesigned with an asynchronous parallel architecture. The first major change was to encapsulate the while loop from Algorithm 2 with an **omp parallel** pragma, and the parallel for loop was changed to use the **nowait** modifier. This removes the implicit barrier at the end of the for loop, allowing individual threads to finish an iteration, generate a new assignment and execute another iteration. Instead of all threads communicating directly with the **ipasir** interface, new clause contributions were added to a queue at the end of each iteration. The master thread (thread ID zero) would add all clauses from the queue to the CDCL sub-problem via the **ipasir** interface, call the CDCL solver, and then use the partial assignment as a seed value. All other threads used the normal PRNG generated seeds using the methods discussed in Chapter 4. Eventually after manual benchmarking and tuning, further optimization greatly improved the performance of the hybrid solver. The first optimization was that only the master thread could contribute random clauses to the

CDCL sub-problem, and it would only do so if there was no local minima detected, or no new UNSAT clauses were included in the last known state of the search. Adding random clauses for each thread only added noise to the hill-climbing procedure, growing the CDCL problem too quickly. Then, in order to force the SLS solver to go in a new search direction instead of retreating to the old local minima, a structure was added to keep track of which variables have been assigned by the CDCL solver. If a variable has been assigned by the last iteration of the CDCL solver, then the *breaks* value is incremented before entering the probSAT flip decision procedure. This metric proved quite effective in forcing the master thread SLS procedure to explore new areas of the search space. The success of this diversification attempt was apparent in how many new clauses were added in each iteration. Previously, it was common for the master thread to not find any new clauses to add for a number of iterations until enough random clauses were added to escape local minima. Finally it became apparent that early in the search it was better for the SLS procedure to perform a smaller number of flips and contribute more clauses due to create a larger base of clauses that are difficult to satisfy, but later in the search you want the SLS threads to search deeper as the CDCL solver takes longer and longer to solve the sub-problems. In order to accommodate this, a dynamic search procedure was adopted. All threads started with a small number of SLS flips, and if any non-master thread finished an iteration without finding a new local minima, the number of flips for the next iteration was doubled.

Performance improved after the above optimizations were applied, and the hybrid solver was able to solve instances, including UNSAT and hard-combinatorial instances, that were otherwise not solved via pure SLS search within the same timeout. Many of these instances were solved before all clauses were added to the CDCL sub-problem. However there were no non-trivial problems where this approach outperformed the standalone CDCL solver.

It is probable that this approach would outperform both SLS and CDCL solvers alone on certain instance classes with the proper tuning and perhaps a different SLS algorithm; probSAT was mostly tuned for random and crafted instances. One of the promising attributes observed while testing the hybrid solver is that on almost all tested instances, the CDCL solver would get stuck or the instance would be solved before all instance clauses were added to the sub-problem.

Another approach that was tested was to have N hybrid solvers where each thread behaves as a master thread and no information was shared between threads. This approach is interesting as it builds on the massively parallel local search concept except with each thread starting with a different seed, and the search direction depending upon the history of the search, it could potentially allow for better scaling/diversification than the traditional concurrent SLS search. One way to think of this procedure is as a tabu-based SLS search where the CDCL sub-problem and learned clauses act as the tabu. While this approach was implemented, it was not tuned and tested extensively as the initial experiments didn't show significant improvement.

5.2 Completeness

All of the approaches discussed above are complete in accordance with the completeness proof from [50]. Essentially since each iteration of the SLS search is finite and each iteration is guaranteed to add a clause (since a random clause is added if no new clauses are found), eventually the CDCL sub-problem will consist of all clauses from the instance. Since the CDCL solver is complete, the hybrid solver is complete.

5.3 Summary

The proposed concurrent hybrid solver did not necessarily outperform both SLS and CDCL standalone solvers on the tested instances. However, there is a justification for further research and tuning of this approach. For instance classes that are generally satisfiable and well-suited to massively parallel SLS search, a low-overhead mechanism to prove unsatisfiability is provided through the asynchronous, lazy clause contribution mechanism proposed above. In [51] it is noted that SMT instances generated by applications such as [54] are difficult and generally expected to be satisfiable. One area where this could be particularly impactful is in embedded applications. For example, the processor cores embedded on FPGAs are typically much less powerful than desktop or workstation processors which would significantly impact performance of CDCL solvers. Embedded applications that benefit from SAT solvers such as [130] could potentially use this approach to accelerate solving in the general case while still being able to prove UNSAT. Additionally the observed behavior of converging towards local minima while ruling out possible solutions would perhaps make for an effective MaxSAT solver.

In the literature there has been little investigation into hybrid SLS/CDCL solvers that are able to take advantage of massively parallel compute resources. This Chapter presented several different approaches to a parallel hybrid solver, with the main contribution being a low-overhead mechanism to retain the benefits of massively parallel local-search while retaining the ability to solve unsatisfiable instances. The contribution here is an approach that allows for a concurrent hybrid solver with low-communication overhead.

Chapter 6

An Effective Concurrent Solver for Industrial Weighted Partial MaxSAT Problems

Weighted Partial MaxSAT (WPMS) is an NP-Hard optimization oriented generalization of the satisfiability problem. As such, it has many important real-world applications in fields such as artificial intelligence[24, 105], planning[70], bio-informatics, weighted constraint satisfaction problems, and other general optimization problems. As with other variants of the satisfiability problem, the real-world applicability of WPMS has led to its inclusion in annual solver competitions, bringing forth rapid innovation in solver algorithms and implementation efficiency. MaxSAT competitions have primarily focused on improving sequential solver algorithms, and this where much of the progress has been made. However, with the stalling of CPU clock speed improvements due to non-linear power consumption and thermal dissipation, chip manufacturers have opted to produce lower frequency chips with more processing cores [110]. Compute intensive problems with efficient parallel algorithms can target traditional super-computing architectures based on clusters of such multi-core CPUs, and may even benefit from hardware acceleration using GPUs or FPGAs.

There have been many attempts to extract concurrency to accelerate solving of satisfiability problems with varying levels of success. Most approaches to extracting concurrency in satisfiability problems have adapted algorithms designed for sequential machines either with a portfolio approach or a search-space partitioning approach. A portfolio solver runs independent solver configurations with different parameters in parallel, hoping that one of the solvers stumbles onto a solution first. While this does often result in linear or even super-linear speedup, it does not guarantee increased performance. Search-space partitioning traditionally assigns branches of the binary decision tree to different solver threads. This approach is difficult to load-balance as traversing branches with the same size search space often takes unequal amounts of time. A fortuitous split can bring a solution to the beginning of a sub-space, resulting in much faster resolution, but other partitioning attempts may decrease performance[60]. Work-sharing approaches for the predominant Conflict Driven Clause Learning (CDCL) SAT solvers also require communication of learned clauses between threads, which can incur significant amounts of overhead. Stochastic Local Search (SLS) based solvers are easier to parallelize as independent solver threads can be seeded differently, however there is no guarantee that these threads will not eventually synchronize, thus performing redundant operations in the same areas of the search-space. In [60] it was suggested that CDCL and Boolean Constraint Propagation based SAT solvers may be inherently difficult to parallelize and that new algorithms and approaches should be considered with parallel architectures in mind.

In this chapter, a parallel local search algorithm for solving WPMS instances is presented. The algorithm has been designed from the ground up to extract both fine-grained and coarse-grained parallelism with the ultimate goal of being deployed on massively parallel systems including compute clusters and FPGAs. Coarse-grained parallelism is exposed through the deterministic seeding of independent local search threads, each of which use a cost function and flip decision metric that should make it unlikely for them to get stuck at the same local minima for extended periods of time. Fine-grained parallelism is exposed through the primary flip heuristic that evaluates the cost of flipping all variables resident in unsatisfied clauses at the current iteration. While this is computationally expensive due to the vast number of unsatisfied clauses and resident variables for large WPMS instances, no data dependencies exist between cost calculations for different variables. This creates a large number of independent tasks that can be processed in parallel at each flip iteration. This type of parallelism is especially well exposed in the FPGA implementation where the process of iterating over clauses and occurrence lists can be heavily pipelined. The proposed algorithm was implemented and tested as a single-threaded program, a multi-threaded OpenMP/MPI hybrid implementation, and as an FPGA port using High Level Synthesis. Despite the simple local search algorithm, performance is competitive with and even superior to solvers submitted to recent MaxSAT competitions on certain instance classes.

6.1 Background and Related Work

6.1.1 Preliminaries

Boolean Satisfiability is the question of whether for a Boolean Formula F across a set of n Boolean variables $x_1, x_2, x_3, ...x_n : \{t, f\}$ there exists a variable assignment such that $F(x_1, .., x_n) \rightarrow t$. This is traditionally represented in Conjunctive Normal Form (CNF) which is a conjunction (logical AND) of clauses where each clause is a disjunction of literals, and each literal asserts the polarity of variable x_i to be positive x_i or negative $\hat{x_i}$.

Satisfiability as posed above is a decision problem with the possible values of satisfiable (SAT) or unsatisfiable (UNSAT). This problem can be generalized to MaxSAT where the goal is to maximize the number of satisfied clauses in an unsatisfiable formula, and Partial MaxSAT where some (hard) clauses must be satisfied and others should be maximized (soft).

This investigation deals with the further generalization of Weighted Partial MaxSAT where each clause in a CNF is assigned a weight and the goal is to minimize the total weight of unsatisfied clauses. There is a maximum possible *top* weight and clauses with this *top* weight are considered hard. All *hard* clauses must be satisfied for the problem to be considered *feasible*, and if the problem is feasible, then the objective is to minimize the weight of unsatisfied clauses for a given solution.

As with SAT, WPMS solvers can either be complete or incomplete. Complete solvers are able to prove that a specified solution is in fact the optimal solution, and incomplete solvers, which are typically local search based will find the best possible score within the allotted execution time but are unable to prove optimality. For WPMS, complete solvers are typically based on branch and bound algorithms, SAT based linear search algorithms with iterative calls to SAT solvers, or they translate the problem to the integer linear programming domain and solve it with a mixed Integer Linear Programming (ILP) solver[6].

6.1.2 Related Work

Arbelaez et. al explored knowledge sharing mechanisms for parallel local search for SAT [7]. They showed performance improvements by sharing knowledge between up to eight SLS solver cores running different local search algorithms. Their four and eight core tests indicate an increase in overall solver performance, but a noticeable amount of overhead is introduced with eight cores. It does not appear that any attempt was made to scale this approach beyond eight cores. The same author was able to show that a parallel SLS approach with no information sharing was scalable up to 512 cores on select instance classes[8].

Entries to the parallel track of recent SAT competitions including Syrup [121], Plingeling, and Treengeling [26] have proven to be effective at solving more instances within the specified timeout than sequential versions of the same engines. Syrup and Plingeling are highly tuned and optimized portfolio approaches with clause-sharing between threads, and Treengeling uses a lookahead based solver to dynamically generate small sub-problems that can be solved quickly by a CDCL solver, an approach known as "Cube and Conquer" [64]. These solvers are extremely effective for SAT, but it is not clear how scalable these approaches are beyond a single machine. Syrup did take advantage of the 24 cores available in the 2017 competition [19] where it won first place in the parallel track [4]. To date, none of the above solvers have been adapted to MaxSAT variants.

Some early work on parallelism in solving Partial MaxSAT is shown in [92] where a set of branch and bound based parallel algorithms were presented that exposed concurrency in the following ways: a two-thread version that explored upper and lower bounds in parallel, a portfolio approach that used different strategies with four-eight threads, and a search-space splitting approach where each thread operated assuming a different local upper bound. While performance was improved over sequential search, speedup was non-linear with the best achieved speedup being 2.57X with eight threads. A distributed MaxSAT solver was presented in [97] and they showed improvements in terms of the number of instances solved up to 16-threads but speedup was notably non-linear.

Sassa and Kanazawa developed an FPGA implementation for Partial Max-2SAT that was presented in[115]. It implements the Dist [29] algorithm and uses parallel, pipelined score evaluators to accelerate the Partial MaxSAT problems with a maximum clause size of two. Speedup of up to 7.74X was reported when using 32-core update blocks. This implementation does not support WPMS or problems with clauses with more than two variables per clause. Many attempts have been made to use FPGAs to accelerate SAT solving with limited success. Typically, these approaches can either scale to solve larger problems with limited speedup, or show great speedup on certain instances, but limited area or performance scalability when solving larger or more difficult problems. An overview of the most successful approaches can be found in [122], [124], and in Chapter 2. Some of the presented FPGA solvers such as [124] and [73] can also be applied to unweighted MaxSAT.

An analog MaxSAT solver was presented in [95] that showed very strong performance on difficult random and combinatorial instances (specifically Ramsey numbers) even when simulated with digital hardware.

6.2 Approach

6.2.1 Insights

The presented solver algorithm and implementation is based on the following core insights, many of which where discovered while attempting to apply a successful FPGA implementation[124] of *probSAT*[15] to MaxSAT and incorporate a weighting strategy for WPMS.

Intensification vs. Diversification

As noted in [79] SLS-based SAT solvers are dependent upon *intensification* and *diversification* where intensification steps improve the objective function and diversification steps move the search out of local minima and into different regions of the search space. They noted along with others[81] that a balance of 80% intensification with 20% diversification empirically produces good results and theoretically lines up with the Pareto principle.

One key insight here is that during the intensification period where a greedy decision is made at every step, it is undesirable to backtrack as this makes it possible to get stuck in a hyper-local minima, with the most extreme example resulting in the same two variables being flipped over and over again. The CCLS[81] solver uses a configuration checking strategy to ensure that only variables whose configurations have changed recently are considered for flipping. In this work a simpler mechanism is proposed wherein only variables that have not been flipped since the last random decision are considered. This ensures that intensification can only proceed unidirectionally with no backtracking and even allows for deterministic diversification as the solver will eventually exhaust all locally optimal decisions and force itself out of local minima by making "poor" decisions. This heuristic heretofore referred to as the *lastRand* heuristic is very effective on some classes of problems, but is harmful in the general case of WPMS benchmarks used in the incomplete track of the 2017 MaxSAT competition. Results for variants of the solver using this heuristic are tagged with "lastRand" in Section 6.5.

Ultimately the presented solver makes random decisions with $p = .1$ with the expectation that the additional diversification enabled by disallowing backtracking during intensification periods will approximate the 80/20 split that has been effective for other SLS solvers[79, 81].

Clause selection vs. Variable Selection

The approach taken by *probSAT* and other SLS SAT algorithms is to choose an unsatisfied clause at random and satisfy that clause by flipping the variable within that clause that maximizes the objective function.

This approach does not work well for WPMS because all clauses are not equally weighted. It is common to satisfy a lightly weighted soft-clause while unsatisfying a hard-clause or many other soft-clauses such that the overall solution cost is greatly increased. Successful local search algorithms for Partial *MaxSAT* and *WPMS*, such as Dist[29] and CCLS[81] consider the flip cost of all candidate variables at each decision step.

Calculating the cost of flipping across all candidate variables greatly accelerates convergence during intensification steps. The key insight here is that the calculation of flip cost over all candidate variables is computationally expensive, but there are no data dependencies between the cost calculations, leaving potential for extraction of fine-grained parallelism.

Probabilistic Approximate Completeness

As proven in[66], an algorithm that allows for an arbitrarily long random walk can be shown to be Probabilistically Approximately Complete (PAC) because as $t \to \infty$, the probability that a solution will be found approaches one. By making an unconditional random decision in the random walk step, the presented algorithm meets the requirements for a PAC algorithm set out by Hoos.

Deterministic SLS Search and Seeding with PRNG

For many applications it is desirable to have deterministic and reproducible results. One of the issues with SLS based solvers is that by definition they rely of stochastic behavior for diversification which makes it difficult to also have reproducable results unless a deterministic seeding procedure is used. The deterministic seeding methods presented in Section 4.1 are applied to different variants of the presented solver and evaluated here. The different seeding methods used include a 64 dimension, 32-bit PCG variant (*PCG32K64*), a 32-bit LFSR with state space divided across cores (*LFSR32*) and a Hamming distance spread method (*spreadHam*). The default is to use the K-dimensional PCG generator, but the LFSR32 and spreadHam variants are appropriately marked in the results section. For making random decisions, the presented solver uses the PCG32 pseudo-random number generator[99], with fast random number generation and good statistical properties. Use of this provides good diversification while maintaining reproducability of results.

Computational Complexity vs. Control Complexity

The primary goal of this solver was to expose concurrency in a way that could be exploited on massively multi-core systems. This resulted in algorithm design decisions and trade-offs that would be otherwise be considered computationally inefficient. The primary example of this is the calculation of minimum cost over all variables within unsatisifed clauses. At first glance, this seems like a prohibitively expensive operation, but without data dependencies between variable flip costs, this operation allows for extraction of fine-grained parallelism.

Age-based tie-breakers

The CCLS[81] solver used the concept of configurations to ensure that variables only considered if related variable assignments had changed since the last flip. This approach has advantages in terms of diversification and avoiding cycles where the same group of variables keep getting flipped over and over again. An attempt was made with the presented solver to use simpler age-based heuristics to keep the control flow of the program relatively simple. Instead of keeping track of configurations, the approach taken here simply keeps track of the last time each variable was flipped, and the last time a random flip was made. This approach is similar to how age is used in the successful Ramp[49] solver, but the actual Ramp mechanism that dynamically adjusts the top weight is not adopted here.

Instead of keeping track of which variables are related to each other, the presented solver first attempts to make the decision with the best score while ignoring variables that have already been flipped in the current intensification period. Ties are resolved based upon $lastFlipped$ values. What results is a solver that generally does not backtrack during periods of intensification, and attempts to make "fresh" decisions when it can't make greedy ones.

Additionally, the independent seeding and age-based tie-breakers make it unlikely for solver threads to fully synchronize and get stuck in the same search space. Even if two threads reached the same local minima and assignment, the different *lastFlipped* values and PRN would help prevent them from running in lockstep.

6.3 Algorithm

The above insights were combined to create the algorithm shown in Algorithm 3 below. The algorithm starts with a random truth assignment across all variables followed by an initialization function that calculates the initial cost and populates the `falseClause` structure. The algorithm then performs a maximum number of flip iterations as part of a local search procedure. With a probability $p = .1$, a clause is chosen using the Pseudo Breadth First Search (PBFS) method discussed in [16] where the current `flip` value is used to index into the `falseClause` structure, instead of picking a clause truly at random. From this clause, instead of flipping a random variable to flip like [81], the least recently flipped variable is chosen. In the general case, all variables residing within unsatisfied clauses are evaluated to find the one with the minimum flip cost where the cost is evaluated as the weighted sum of clauses that would be unsatisfied by flipping that variable minus the weighted sum of clauses that would be satisfied. The aim is to minimize the cost, and a negative flip cost would result in a better overall score, thus the variable with the minimum cost is selected. In the case of a tie, the variable that was least recently flipped is chosen. In the case that there is a tie and multiple tied candidates have not been flipped yet, a random decision is made between them. Additionally, the novel heuristic that prevents backtracking after a random decision has proven effective on some instance classes and detrimental on others. Benchmarks using this *lastRand* heuristic are tagged appropriately in Section 6.5.

Algorithm 3 Algorithm for presented solver

```
$\alpha \leftarrow$ random truth assignment
$cost \leftarrow cost(F, \alpha)$
$falseClause \leftarrow$ unsatisfied clauses
for $flip = 0$ to $maxFlips$ do
    if $true$ with probability p then
        Choose $bestVar$ with oldest flip from random clause
        $lastRand \leftarrow flip$
    else
        $cost_v \leftarrow \sum w_c(break) - \sum w_c(make)\ \forall v \in falseClause | (lastFlip_v < lastRand)$
        $bestVar \leftarrow \min(cost_v)$
        if $\sharp \min(cost_v) > 1$ then
            $bestVar \leftarrow \min(lastFlip_v)$
        end if
    end if
    $lastFlip_v \leftarrow flip$
    $falseClause \leftarrow$ UNSAT clauses after flip
    $cost \leftarrow \sum w_c \forall c \in falseClause$
    if $cost < bestCost$ then
        $\alpha\star \leftarrow \alpha$
        $bestCost \leftarrow cost$
    end if
end for
```

6.4 Methodology

Algorithm 3 was implemented in C++ using OpenMP and MPI to enable concurrent operation on a distributed compute cluster.

In order to judge whether this parallel WPMS solver is effective on real-world problems, a set of benchmarks from the MaxSAT2016 and MaxSAT2017 competitions were run with the same 300s timeout used during the competition. Much of the focus in benchmark selection was on the Industrial track as a major goal is to impact performance on real-world application benchmarks. When considering rankings, the same convention as the competition is used where the first priority is finding the best score, and the second priority is finding it in the least amount of time.

The machines used in the cluster housed Intel(R) Xeon(R) CPU E5-2683 v4 processors running at 2.10GHz with 32 cores. Results from the presented solver are compared to results from the 2016 and 2017 MaxSAT competitions where benchmarks were run on the StarExec cluster where each node housed Intel(R) Xeon(R) CPU E5-2609 processors running a 2.40GHz. While the two CPUs are different architectures running at different clock speeds, there is evidence that the disadvantage in clock speed for the newer processor is balanced out by architectural enhancements[53]. If anything, the presented solver is being put at a disadvantage as it is being compared to *all* of the incomplete solvers from the respective competitions, effectively building a well diversified parallel portfolio of the top solvers.

6.4.1 Initial Implementation

A **wMaxSatWorker** class was created to encapsulate the main solver algorithm. The constructor makes thread-local copies of the clause database, occurence table, and clause-weight structure in order to make sure that memory accesses don't need to go across nodes. A for loop in the main function generates worker threads using the **omp parallel for** construct. The initial assignment for each search thread advances the PRN by $64 * thread\_id$ and then assigns each 32-bit random integer to a 32-bit block of the assignment space. Then the `solve` function is invoked and it runs until `flip` == `maxFlips` or the program is killed due to a timeout. When the timeout occurs, the best score along with the flip index and timestamp of that score is printed out. The work-sharing version of this solver allocates different subsets of candidate variables to each thread. The individual threads calculate the flip cost of their assigned variables, and select the best candidate, and finally a sequential process selects the best candidate amongst those chosen by the different threads.

The initial implementation used only OpenMP and didn't scale beyond a single node which supported 32 threads on the target cluster. Before calculating the cost of flipping each variable, the candidate variables were enumerated by iterating over the `falseClause` structure and storing the contained literals in an unordered set. This data structure was chosen to prevent redundant calculations of flip cost.

However, using the unordered set had some unintended consequences both positive and negative. The negative consequence was easily seen when profiling the code, the hash functions used to determine uniqueness dominated execution time. The positive consequence wasn't realized until further optimization attempts to use faster data structures such as **std::vector** for the intermediate data structure. Despite the redundant flip calculations, storing candidate variables in a vector nearly double the flips/second; this somewhat makes sense considering the computational complexity of the hashing functions. The less intuitive result was that the solver did not converge as quickly when using a vector as the intermediate store, despite the increase in flips/second. After in-depth analysis to figure out where exactly the two approaches made different decisions, it became apparent that the unordered

Table 6.1: Fine-Grained Work-sharing

Instance	1T	2T	4T	8T	16T	32 T	MaxSAT2016		
							Best	Worst	AVG
ar-1.wcnf	25254,101	25114,285	25062,199	25062,114	**25062,63**		25084,5s	TO	25191
1401.wcsp.dir.wcnf	473107,38	473105,281	473105,202	473107,32	473107,33	473107,33	**465097,218**	TO	485503
1403.wcsp.dir.wcnf	471263,47	470261,268	470261,178	470261,261	471264,35	471263,35	**465239,148**	TO	484257
1405.wcsp.dir.wcnf	465458,145	465458,88	**465458,64**	465458,73	465458,83	495438,228	469420,295	TO	492819
1407.wcsp.dir.wcnf	467616,148	467603,269	**467603,181**	467603,229	467603,271	501627,190.604	475568,294.68	TO	489620

set offered a well distributed third-level tie-breaker for flip decisions. Essentially, there are scenarios where multiple flip candidates share the same flip cost and the same `lastFlip` value of zero if neither had ever been flipped. This was a scenario that was not anticipated when designing the algorithm, but is nonetheless quite common in early stages of the search.

Another issue that became apparent with the work-sharing version of the algorithm is that results were not consistent when changing the number of worker threads, and often got worse as the number of threads increased, again despite an increase in flips/second. This again came down to the second-level tie-breaker. When there are multiple variables with the same flip cost, and none of them have been flipped yet, the one that is flipped is the one that appears first in the candidate flip data structure. When this data structure is split amongst worker threads, this ordering is different per thread, resulting in different decisions being made early in the search and ultimately different results. While several different methods were tried to resolve this issue, the most performant solution was to use a vector to store the candidate variables and before entering the parallel region where cost is calculated a random index value between 0 and `numVars` was chosen as a target, and the variable closest to this value would be chosen in case of a second-level tie. This removed the hash function bottleneck while allowing for consistent performance in terms of number of flips to reach a solution when increasing worker threads.

Performance results for this initial implementation are presented in Tables 6.1,6.2 below.

Table 6.1 shows the results of using work-sharing constructs to extract fine-grained paral-

Table 6.2: Independently Seeded Solver Threads

Instance	1T	16T	128T
1401.wcsp.dir.wcnf	476107,20	474111,187	471104,182
1403.wcsp.dir.wcnf	NA	472267,249	470261,110
1405.wcsp.dir.wcnf	NA	471443,60	470449,210
1407.wcsp.dir.wcnf	NA	470606,294	**470602,52**

lelism within the solver from 1-32 threads. The MaxSAT 2016 column represents the winning score and time from the 2016 MaxSAT competition, and the optimal solution is one that was proved optimal in that competition. It is notable that for the ar-1 relational inference instance, all versions using four threads or more were able to find an optimal solution, something none of the 2016 competition solvers were able to do within the 300s timeout. For the `wcsp.dir` instances, the optimal number of worker threads was four. With four threads, the presented solver found a better solution for the 1405 and 1407 instances than any of the MaxSAT2016 incomplete solvers.

Table 6.2 shows some results from benchmarking the coarse-grained parallel solver that seeded each thread differently and uses the unordered set data structure for storing candidate variables. Performance typically increases with this approach, but there is significant overhead in spawning the worker threads that prevents the full advantage of this approach from being displayed within the 300s timeout. Additionally the 128T column likely represents the results of 32 threads running on a single node as the target cluster does not support a virtual shared memory architecture.

In both tables, boldfaced text highlights the best results.

6.4.2 Further Optimizations

MPI and NumaCtl to scale

The initial pure OpenMP implementation of this application limited the implementation to a maximum of 32 processing cores on the target cluster. Additionally, there were performance anomalies when using shared memory for the clause database. Performance was consistent when using thread-private memory structures for the clause database, but this significantly increased memory requirements. An attempt was made to port the existing code to MPI to enable greater scalability, and initially it seemed like MPI was a more natural implementation for the non work-sharing version of the WPMS solver. However running N full executable threads on a single node with MPI did not scale very well and was significantly slower than the OpenMP implementation, likely due to the fact that each MPI process was trying to load and parse instance files off the same network file system.

Ultimately, a hybrid architecture was developed that would run one MPI task per node and each MPI task could spawn up to maximum number of threads available on each node using OpenMP. This architecture was much more scalable across nodes, however there were still some performance anomalies when using large shared memory structures, and the performance didn't scale linearly from 4-32 threads on a single node even when considering the decreased maximum clock speed for heavily threaded workloads. It turned out that threads were not necessarily being assigned the same socket, which introduced significant overhead accessing shared memory resources. The solution was to spawn two MPI tasks per node and use the `numactl` utility to assign socket affinity to the MPI tasks as shown in [109]. Each MPI task spawned 16 OpenMP threads. With each MPI task sharing socket-local memory among its OpenMP threads, flips/second per thread remains fairly constant. Another feature of this hybrid architecture that could benefit users on shared compute clusters is that it enables the user to use cores that would otherwise be inaccessible due to fragmentation. Instead of waiting for completely free nodes, machines with any number of cores free could run an MPI task that spawns the appropriate number of threads.

Implementation Optimization

Further profiling of the single-threaded code showed significant bottlenecks in the hash function for the unordered set as well as the flip cost calculation. This version of the implementation was also switched to using vectors as the candidate variable data structure, and any time a second-level tie was encountered, a random number was generated to choose between two candidate variables. The amount of calculation necessary was greatly reduced by leveraging break caching and XOR techniques for keeping track of critical variables as described in [16]. With these optimizations included, the benefit of fine-grained work-sharing parallelism when calculating flip cost was mostly wiped out as the overhead of spawning and synchronizing OpenMP threads overwhelmed any speedup. Additionally, the third-level tie breaker mechanism was simplified, if there is a scenario where two variables share the same flip cost and flip age (neither has been flipped yet), a simple coin toss mechanism requests a random selection of 0 or 1.

Finally more optimizations were done using the significantly more difficult MaxSAT2017 benchmark set that found that some of the heuristics that worked well on the ar-1 and other instances do not work well in general on other instances. For example, the heuristic that disallowed flipping variables that had already been flipped since the last random decision seemed to hurt performance on some instance classes and help it on others.

Some results with the coarse-grained solver following these optimizations are shown in Table 6.4.

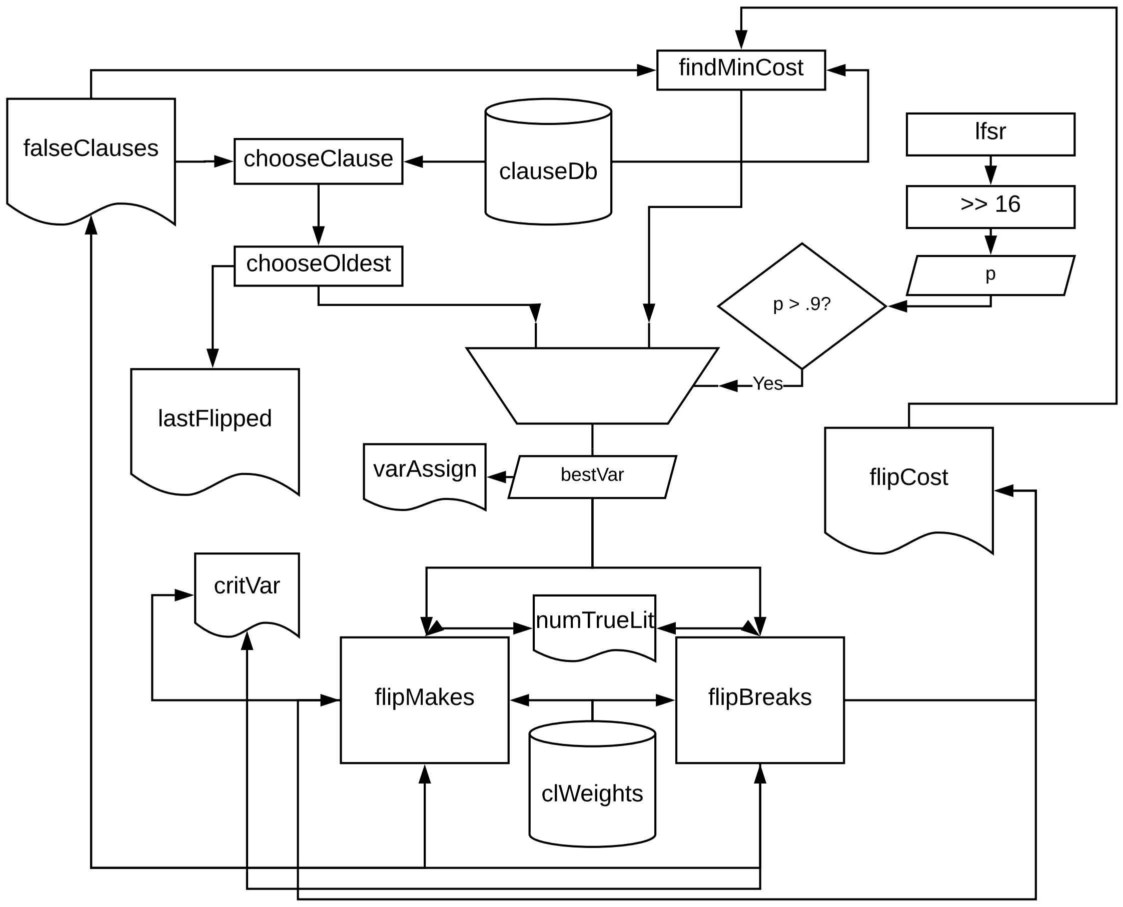


Figure 6.1: PickAndFlip Module Block Diagram

6.4.3 FPGA Implementation

One of the major motivating factors of this work was the ability to use hardware accelerators for satisfiability based problems. Performance of the MPI/OpenMP software solver is quite competitive on some instance classes, however, an FPGA implementation is able to increase compute density, power efficiency, and enable embedded use of the accelerator. One could even foresee the use of such an accelerator in a hybrid approach similar to how the Ramp[49] solver is used as part of the Maxroster[127] solver.

The HLS implementation of the many-core probSAT solver from Chapter 3 was used as a based with additional memory resources needed to store the cached flip cost of each variable and the clause weights. The parser for the WPMS solver was modified to print a header file to correctly initialize the memory structures in HLS. The algorithm is largely identical to the one presented in Algorithm 3 with some of the data types and structures moved around for more efficient hardware implementation. Any time a loop went over variables in a clause, or occurrences in the occurrence table, an `HLS_PIPELINE` pragma was inserted. This is significant because this pipeline processing exposes fine-grained parallelism that couldn't be efficiently extracted with OpenMP constructs in software. Ultimately this meant that one the WPMS cores on a weighted Max2SAT instance is able to achieve almost $\frac{2}{3}$ the performance of a single Xeon core in terms of flips/sec despite running at more one order of magnitude lower clock speed. This shows again how the extra per-flip computations in WPMS is perhaps better suited to hardware implementation than regular SAT where each FPGA core achieved around $\frac{1}{10}$ the performance of a Xeon core. In several of the pipelined loops there were carried loop dependencies that prevented full pipelining on every clock cycle. Some dependencies on the `critVar` array were false dependencies, and adding the appropriate pragmas removed the bottleneck. Other depdencies on the `flipCost` structure were true carried dependencies and the iteration interval of the pipelines were limited to nine clock cycles due to the read-modify-write access on `flipCost` and each loop along with the limited ports on the BRAM. With more optimized memory structures it is possible for even more performance to be extracted from this fine-grained parallelism. Other optimizations for efficient hardware implementation, similar to the work presented in Section 3.3, include changing out PRNG calls for a simple 16-bit LFSR and replacing floating point probability calculations with fixed-point equivalents with Xilinx's **ap_fixed** classes. The cointoss mechanism for third-level tie-breakers was modified to behave like the jump-ball mechanism in basketball, it is a one-bit counter that oscillates between zero and one each time a tie is broken. If the value is zero, then the current best candidate is kept, otherwise the new candidate is chosen.

Table 6.3: WPMS Core Utilization

Instance	BRAM	DSP	FFs	LUTs
t7g3-9999.spn	31	0	2270	4560
1401.wcsp.log	603	2	2958	4643
Grids_30(UAI Conversion)	64	0	2315	4591

The core generated from the HLS code is generated and exported as a design checkpoint file and instantiated in a top-level verilog file. A generate statement instantiates the specified number of cores, and a lookup table allocates LFSR seeds based on precomputed values that spread the state space of the 16-bit LFSR across cores. A set of 1-bit wide RAM modules were instantiated using the Xilinx Language templates in order to store variable assignments and current best assignment values. These were connected to the memory interfaces synthesized by the HLS compiler.

A block diagram of the variable selection and flip (pickAndFlipHW) routine is shown in 6.1. The resource utilization of this core is shown in Table 6.3 for several tested instances.

Scaling to Large Cloud Hosted FPGAs

One of the barriers for adoption of hardware acceleration is the relative cost and complexity of purchasing, maintaining, and developing for custom hardware. One option is to host accelerators in the cloud, Amazon provides such an infrastructure with their hosted F1 instances[5]. These instances support the use of up to eight Xilinx XCVU9P UltraScale+ FPGAs. In this section, it is illustrated how the presented concurrent WPMS solver might scale to such an instance.

Figure 6.2 uses the resource utilization of the Grids_30 instance to show how the number of cores might scale from the tested XC7VX690T device to the Amazon F1 instance with up

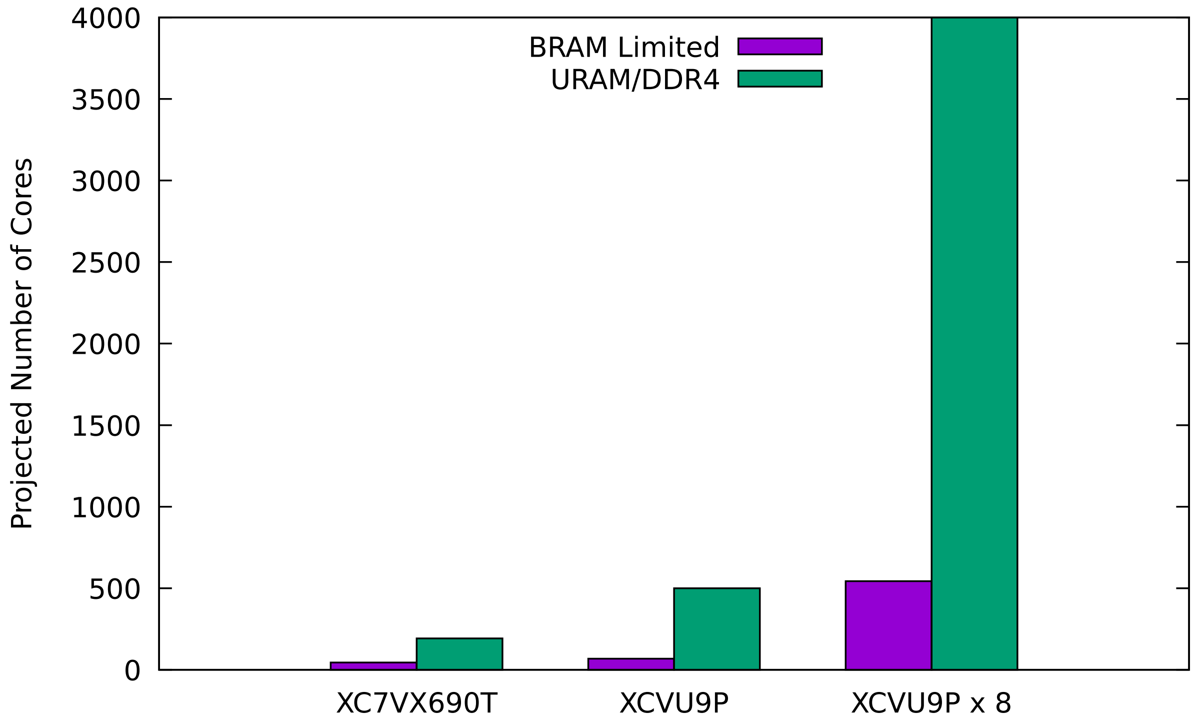


Figure 6.2: Core Scaling of Grids_30 UAI Instance to F1 Instance

to eight XCU9P devices. Figure 6.3 shows how performance might scale in terms of flips per second for the BRAM limited instance considering the HLS compiler for the Virtex 7 device was able to achieve timing closure at 200MHz compared to 333MHz for the UltraScale+ device.

6.5 Results

The solver algorithm and distributed processing seemed particularly well-suited to the WCSP translations of the Satellite data optimization problems presented in [22]. On the `1401-1407` variants of the `wcsp.dir` problems, the presented solver found better solutions than any of the incomplete solvers from the MaxSAT2016 competition as shown in Table 6.4. The `[1401-1405].wcsp.log.wcnf` log instances were used to test incomplete solvers in both the

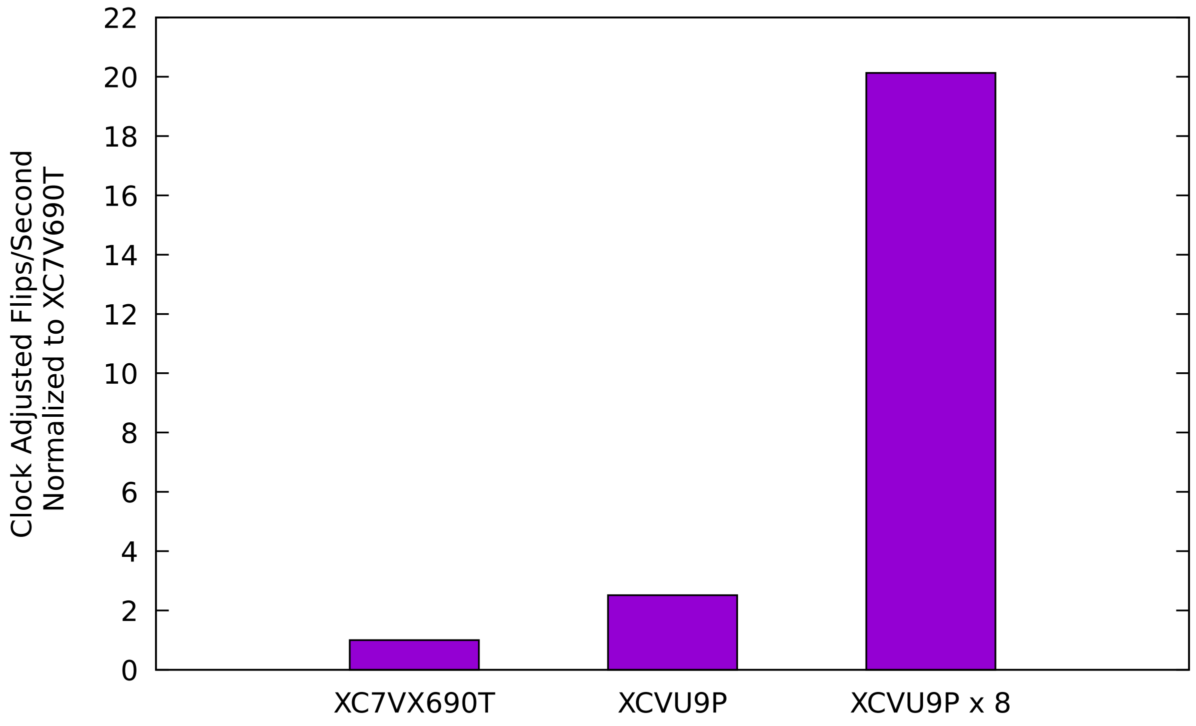


Figure 6.3: Performance Scaling of Grids_30 UAI Instance to F1 Instance

2016 and 2017 MaxSAT competitions and comparison results from these benchmarks are shown in Table 6.5. Note that the `wcsp.dir` instances with results shown in Table 6.4 and the `wcsp.log` instances with results shown in Table 6.5 are distinct instance classes with different optimum solutions. Once again, the presented solver outperformed all entrants to the 2016 MaxSAT competition when used with 64-128 threads. The Maxroster[127] solver dominated the 2017 competition and outperformed the presented solver on all instances here. However, the performance of the Maxroster solver was an outlier, and the presented solver found a better solution with 64-128 threads than any other 2017 solvers on these instances. It is also worth noting that the Maxroster solver uses the Ramp[49] local search solver to generate the initial upper bound. The presented solver could replace Ramp, or even adopt the Ramp heuristics with the massively parallel infrastructure and further increase the overall performance of the Maxroster solver.

Table 6.4: Independently Seeded Solver Threads: Spot5 **DIR** Instances

Instance	16T	32T	64T (PCG)	128T(LFSR32)	MaxSAT 2016		
					Best	Worst	AVG
1401.wcsp.dir.wcnf	467096,274	467096,119	465102,246	**461105,119.34**	465097,218	TO	485503
1403.wcsp.dir.wcnf	465241,89	465240,290	**463241,241**	465242,289	465239,148	TO	484257
1405.wcsp.dir.wcnf	465420,258	465420,270	465420,258	**463419,178**	469420,295	TO	492819
1407.wcsp.dir.wcnf	471581,281	467580,282	467580,289	**465582,167**	475568,295	TO	489620

Table 6.5: Independently Seeded Solver Threads: Spot5 **Log** Instances

Instance	64T (LFSR32)	128T (LFSR32)	MaxSAT 2016			MaxSAT 2017		
			Best	Worst	AVG	Best	Worst	AVG
1401.wcsp.log.wcnf	461100,107	461100,102	463099,196	TO	491009	**459111**	496107	476857
1403.wcsp.log.wcnf	461243,184	461242,30	463242,224	TO	492156	**459270**	511272	481260
1405.wcsp.log.wcnf	461425,293	461425,277	462443,200	TO	492013	**459454**	509452	484072

Another strong point of performance are the Relational Inference instances that were used to test incomplete solvers in the 2016 MaxSAT competition. The results are shown in Table 6.6. The presented solver with 64 threads, seeded by the LFSR32 and using the lastRand heuristic shined on the `ar-1` instance, finding the optimum value in 5.57 seconds, whereas none of the 2016 competition entrants were able to find the optimal solution within a 300s timeout. Performance on the `ar-3` instance was also excellent with the 32 thread variant using spreadHam seeding and the lastRand heuristic found the optimal solution with a 375X speedup over the best incomplete solver in the 2016 competition.

Table 6.6: Independently Seeded Solver Threads: Relational Inference

Solver Variant	ar-1	ar-2	ar-3	rc-1	rc-2	rc-3
32T (spreadHam)	25080,1.18	395196,291.51	43814,3.31	9816,4.38	10126,4.55	9816,4.34
32T (spreadHam,lastRand)	25062,7.51	395390,222.08	**43814,0.62**	10338,4.74	11006,4.50	10338,4.16
64T (LFSR32)	25080,1.10	395192,273.60	43814,6.88	9816,4.26	10126,4.16	9816,4.25
64T (LFSR32,lastRand)	**25062,5.57**	395390,233.56	43814,11.19	10338,4.96	11006,4.27	10338,4.82
MaxSAT 2016	25084,4.68	**394844,59.53**	43814,233	**5722,24.64**	**5722,251.64**	**5722,264.51**

Tests were run to evaluate performance of the solver on the very difficult set of benchmarks used in the incomplete track of the 2017 MaxSAT competition. The presented solver was competitive on the `Spot5`, `maxcut`, and `rna_alignment` instances. The 64 thread solver would have tied for first place on the maxcut instances, having found the optimum value in all

Table 6.7: MaxSAT 2017 Incomplete Track Comparison

Solver Variant	Instances Solved
1T (PCG)	35
16T (spreadHam)	68
32T (PCG)	75
32T (spreadHam)	73
32T (spreadHam,lastRand)	49
64T (spreadHam)	70
maxroster (MaxSAT2017)	149
Open-WBO-LSU (MaxSAT2017)	92

cases. The Ramp solver is the only solver that found optimal values for all MaxCUT instances in both the 2016 and 2017 (as a component of maxroster) competitions. A comparison of solver variants with different seeding procedures is shown in Table 6.7 where performance is measured in terms of instances solved. Based on the 1T performance results it is clear that the core solver is not very competitive aside from the instance classes presented above. Performance does seem to scale well up to 32T cores, but drops off a bit with 64 cores.

6.6 Summary

This chapter presented a new parallel solver and hardware accelerator for WPMS problems with the potential to exploit massive amounts of concurrency. Results on hard problems from recent MaxSAT competitions were presented along with scaling potential of the presented hardware accelerator. The simple local search algorithm presented here was not really competitive as a single-threaded solver, but scaled well, solving significantly more instances up to 32 threads in general and up to 128 threads depending on the instance class and type of deterministic seeding used. In particular quality of results scaled best on the `Spot5` WCSP problems derived from [22], for this class of problems, either the score or time to reach the score improved up to 128 threads. It also is apparent from the results that the different seeding methods and heuristics have different performance and scaling profiles on different instance classes.

The contribution presented here is a concurrent solver for WPMS problems that is efficiently implemented in both multi-threaded software and FPGA hardware. There do not seem to be other WPMS hardware solvers found in the literature, [115] implemented a partial Max2SAT solver for unweighted problems with maximum clause size of two, a limitation not shared by the presented solver.

Chapter 7

Case Study: Concurrent WPMS Solver Applied to Probabilistic Inference

One potential application area for WPMS is in solving probabilistic inference problems. These problems encode Probabilistic Graphical Models (PGMs) such as Bayesian or Markov networks into an optimization problem that attempts to maximize the a posteriori probability of a random variable given evidence of some other random variable. This attempt to find a "most likely" explanation for unknown variables can be applied to many real-world problems such as image segmentation. While probabilistic inference problems can be solved directly in their native domain with branch and bound algorithms such as daoopt[102], and have spawned competitions as part of the Uncertainty in AI Conference[55], the constant advancement in performance for SAT and MaxSAT solvers motivated translations of these problems to MaxSAT and WPMS as in [11, 105].

There are examples of successful distributed solvers for inference problems in the literature such as [103]. High-performance FPGA solvers for inference also exist, but they do have limitations. For example the work presented in [10] offers great performance but only fits 29 nodes on a single FPGA. While their architecture is designed to scale with multi-FPGA systems, using it for more difficult problems with thousands of variables seems infeasible.

In [32] a high performance solver is presented, but the design is highly-tailored for a specific application with certain structural qualities in the graph model, it isn't clear that this approach would be appropriate for a general purpose inference solver.

The conversion to WPMS is particularly appealing for FPGA implementation because it translates from the floating-point probability values in the probabilistic inference domain to integer weights that can be more efficiently computed in hardware. A solver algorithm and implementation was presented in Chapter 6 that is able to take advantage of fine-grained parallelism and coarse-grained parallelism in the WPMS problem.

7.1 Methodology

The translator code to convert probabilistic inference instances from the UAI[55] format is taken from [68]. In this conversion process, each random variable in the Markov random field (MRF) has a set of possible states. A Boolean variable is created for each possible state of each variable, and is set to true if that variable is in that state. The function tables that represent the factors are encoded into soft clauses where the probabilities of each line in the function table translates to the weight of the respective clause. The hard clauses are generated to ensure that for each variable, at least one state is active, thus ensuring that there is a full assignment of variable states for any valid solution. This encoding leaves open the possibility that a Markov variable might be assigned two active states, but in practice this isn't an issue since such an assignment would not be locally optimum.

The focus of this case study is to compare the performance of different SLS solvers on the generated instances. The experiments presented below were performed on a compute cluster equipped with two Intel Xeon E5-2683 CPUs per node for a total of 32 cores. The single-threaded solvers are actually likely to be running at a clock speed advantage as the tested

CPU supports turbo-boost up to 3.0GHz. Solvers were run with a 20s timeout to mimic the fast track in the UAI2014 competition. Multiple versions using different deterministic seed methods were tested including the variants presented in Chapter 6 as well as a new variant that adopted some parameters from the Ramp[49] solver.

7.2 Results

A performance comparison between the parallel solver presented in Chapter 6 and several WPMS solvers submitted to recent MaxSAT competitions is presented here. These include Ramp[49], CCLS[81], and Open-WBO[90]. Note that while Open-WBO is a complete solver, it does print out intermediate results which were used for this comparison. It was not able to find a provably optimal solution to any of these problems within the allotted time.

Table 7.1 shows the performance of the various WPMS solvers on the probabilistic inference instances. For each instance (named as they were in the UAI competition), the table shows the lowest cost (the sum of all unsatisfied clauses) for each solver. The solver that finds the lowest cost solution is considered the winner, for each instance the winners score is presented in **bold**. The instances shown here were selected with a few representative instances from each class of inference models that were successfully converted. A seperate `wpms_verify` application was written to double-check the score, and verify that no hard-clauses were left unsatisfied. Note that some of the results for the 64-thread version with spreadHam seeding were disqualified due to several hard clauses being UNSAT and are marked as NA in Table 7.1. This happened because the converter did not appropriately assign the weight for the hard clauses as the sum of all soft clauses, allowing some hard-clauses to remain unsatisfied even with a very low cost solution. The LFSR32 seeded solver variants presented in Table 7.1 were modified to automatically correct the hard clause weight value as the instance is being parsed to avoid this situation.

From Table 7.1 it can be seen that the parallel WPMS solver offers competitive performance on these instances. The 64-thread and 128-thread versions of the algorithm found the best cost solution on three out of eight instances. The modified version which adopted heuristics from the Ramp solver including the actual Ramp functionality which drops the hard clause weight to the current best cost value, as well as the more aggressive diversification (random decision with 40% probability) performed extremely well, finding the best cost solution on four out of eight instances, easily beating the single-threaded Ramp solver. The single-threaded Ramp solver taken from [93] performed well on the instances it could find a solution for, but was unable to find any solution for the larger WPMS problems. It appears that the version of Ramp used as part of Maxroster[127] in the 2017 competition was modified to support larger problems.

Table 7.1: WPMS Solver performance on Probabilistic Inference Instances (20s timeout)

Instance	64T (SH)	128T (LFSR32)	Ramp	64T (LFSR32,Ramp heuristics)	CCLS2015	Open-WBO
Segmentation_15	**2198489**	**2198489**	NA	**2198489**	2578949	3521121
Grids_30	99861085	97892971	90346675	89165126	117410415	**88026726**
wcsp_11	260427	259510	257664	**255822**	269657	257664
wcsp_12	187147	187171	174821	174797	193526	**174635**
wcsp_17	NA	464689654	NA	455916197	148074	207355
wcsp_16	NA	170259816	NA	119799351	161764	**35105**
Segmentation_20	**1529389**	**1529389**	NA	**1529389**	2180235	3768228
Segmentation_14	**66473428**	**66473428**	NA	**66473428**	76811579	66473428

After conversion of the WPMS solution back to the inference domain, the WPMS solvers generally got lower (worse) MAP scores than the domain specific solver daoopt[102] which had won the Gold medal in the 2012 UAI competition and silver in the 2014 competition. The exceptions were on instances such as `Segmentation_14` and `Segmentation_20` where both found the same solution.

A 32-core hardware accelerated solver for the Grids_30 instance was also implemented on a Xilinx Virtex-7 690T FPGA running at 200MHz. Resource utilization for this implementation is shown in Table 7.2. The best cost solution for this solver within 20s was 97328778 which puts it in the middle of the pack for the tested solvers. It's worth noting however, that the estimated power consumption for this solver is only 1.19W, two orders of magnitude less than the Xeon processor used to test the software solver. The FPGA solver uses a 16-bit LFSR to seed the cores and generate random values for making decisions. A more detailed discussion of the WPMS FPGA solver algorithm and implementation is presented in Section 6.4.3.

Table 7.2: Grids_30 32-core Hardware Solver

Resource Type	Number Used	Utilization
BRAM	1100.5	75%
SLICE LUTS	77049	18%
SLICES FFS	82366	10%

A layout view of the implemented FPGA solver is shown in Figure 7.1.

7.3 Summary

Probabilistic inference is a compelling application domain with a straightforward conversion to WPMS. While none of the tested WPMS solvers are competitive with the domain specific daoopt solver, it could serve as an interesting set of benchmarks for WPMS solvers as the tested competition solvers do not seem well tuned for them.

The presented parallel solver did outperform other WPMS solvers in the general case, especially when combined with some of the heuristics from the Ramp solver where the best solution was found on four out of eight instances.

The contribution presented in this chapter was an evaluation of the parallel WPMS solver as it applies to problems converted from Probabilistic Inference problems from the UAI

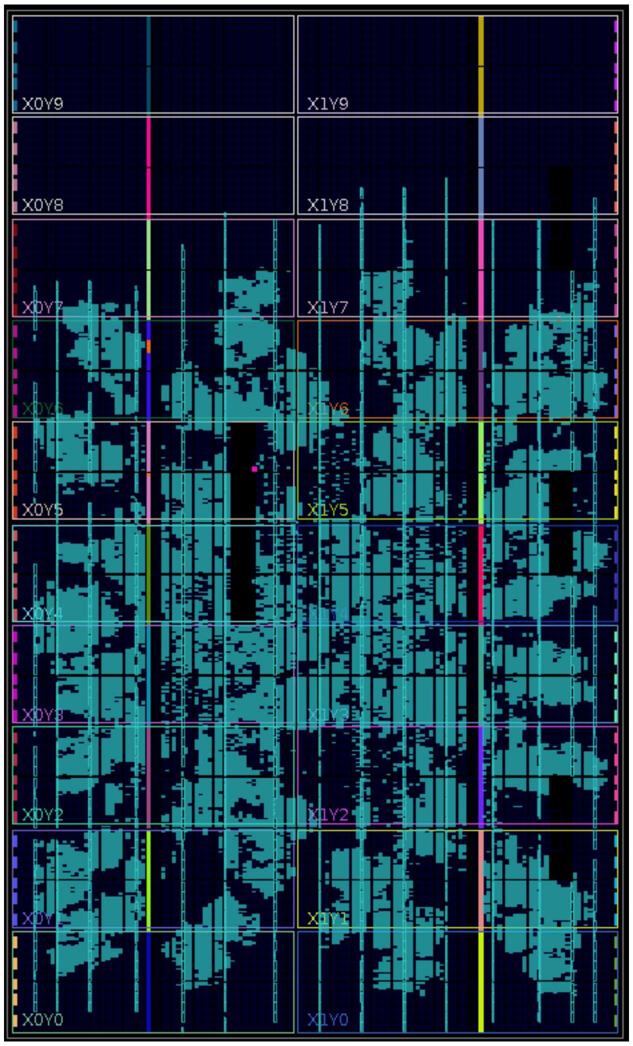


Figure 7.1: Grids_30 HW Solver Layout

competition. A FPGA implementation of this solver was also presented. The low-power consumption of the FPGA implementation makes it an interesting option for embedded applications of inference problems. This appears to be the first FPGA WPMS solver presented in the literature, and thus also the first applied to probabilistic inference problems using WPMS.

Chapter 8

Conclusions and Future Work

The overarching goal of this work was to advance the state of concurrency in satisfiability. To this end, the document started with a thorough survey, discussion, and analysis of the current state of SAT solvers and algorithms and the attempts to exploit concurrency in SAT using multi-core general purpose compute architectures as well as specialized hardware accelerators. The analysis and insights presented in Chapter 2 informed a path forward to further advance the state of concurrency in SAT.

The first insight was that following [8], not much work had been done to advance parallel local search as a viable approach to accelerating satisfiability. Kanazawa et. al had shown some success in accelerating WalkSAT by extracting fine-grained parallelism through parallel clause evaluation[71, 72, 73, 74], and they extracted some coarse-grained parallelism by running enough independent parallel search threads to keep their pipeline saturated. The contribution that followed this insight was a many-core SAT accelerator with an efficient, high-density FPGA implementation as well as a performant OpenMP/MPI implementation intended for general purpose compute clusters. This contribution was presented in detail in Chapter 3. This solver showed speedup, sometimes super-linear speedup of up to four orders of magnitude over the sequential probSAT implementation as well as up to five orders of magnitude over MiniSAT on relatively difficult, crafted instances in the `sgen1` class.

It was noted in Chapter 3 that the simple 16-bit LFSR used for seeding the SLS cores and providing stochastic behavior for random decisions performed quite well, especially after optimizations were made to the seeding procedure to evenly spread the LFSR state space across the cores. Evenly spreading the LFSR16 state space across the cores on the `sgen220` instance for example presented an astounding 62000X speedup. This observation, coupled with the poor multi-threaded performance of the **libc rand()** function motivated an investigation into the importance of statistical soundness of PRNGs used in SLS SAT solvers and investigation into deterministic seeding methods, including a look into a deterministic seeding method that can guarantee minimum Hamming distances between seeds while distributing assignments across the range of possible Hamming weights. The results from this investigation are presented in Chapter 4 with the primary contributions being a simple Hamming-distance based initialization scheme with guarantees about the minimum Hamming distance between solver cores and experiments showing that simple and efficient LFSR based PRNGs in hardware implementations are not necessarily a losing ground in terms of overall solver performance.

One issue with SLS solvers in general is that they are inherently incomplete, and many real-world applications of satisfiability require the ability to prove an instance to be unsatisfiable. Additionally there are many classes of application instances where SLS solvers, even when well distributed across the search space are not going to be competitive with complete CDCL based solvers. To address these shortcoming, works such as [50] and [77] developed hybrid solvers with iterative procedures that allow SLS and CDCL components to collaborate during the search. Aside from [20] where the combined OpenCL GPU SLS search and MiniSAT combination were 4X slower than an unmodified version of MiniSAT, not much work has been published regarding parallel hybrid solvers. In Chapter 5 multiple approaches to incorporating a complete solver with the massively parallel local search solver from Chapter 3 are considered. While it doesn't appear that any of those approaches give performance benefits over simply running independent SLS and CDCL solvers on the tested instances, the presented solver approaches are provably complete. Thus the ultimate contribution from Chapter 5 is a hybrid approach that uses asynchronous, lazy communication of local minima to enable massively parallel search without losing the ability to prove unsatisfiability.

One of the takeaways from Chapter 2 is that generalizations of satisfiability such as MaxSAT and WPMS are less explored in terms of scalable concurrent algorithms in general and there have been few attempts at hardware acceleration in particular. The most relevant works include [115] where the Dist algorithm for Partial MaxSAT was implemented with fine-grained parallelism being exposed in the form of parallel clause evaluation and score updates while [95] presented an analog MaxSAT solver that showed strong performance on "very-hard" MaxSAT instances even though the algorithm was being run on a digital computer instead of analog hardware. The contribution presented in Chapter 6 is a scalable parallel local search solver that shows strong performance on certain classes of instances including MaxCUT and Spot5[22] instances that are typically quite difficult for complete solvers. A simple greedy algorithm with age-based tie-breakers was developed and parallelized within an OpenMP/MPI framework as well as a hardware implementation based on the infrastructure presented in Chapter 3. Fine-grained parallelism is exposed here via clause and occurrence processing pipelines, and performance in terms of flips/sec of a single core is within $\frac{2}{3}$ that of a Xeon core despite a two order of magnitude handicap in clock speed on older FPGA hardware. Performance projections onto modern UltraScale+ hardware in terms of number of cores and flips/second is presented to adjust for greater capacity and clock speed available on these devices. A single Amazon F1[5] instance could potentially host up to 4000 of these cores. There are no references in the literature to a hardware accelerator for WPMS and unlike the Max2SAT solver presented in [115] the presented architecture supports instances with more than two variables per clause.

Finally a case study applied the WPMS solver presented in Chapter 6 to instances generated from probabilistic inference benchmarks. The presented solver performs competitively on such instances and is able to find lower scores than the tested WPMS solvers on select instances. The parallel WPMS solver does not outperform the domain specific daoopt[102] solver that was effectively parallelized in [104]. However, that complex software algorithm is is dependent on floating point performance, and it may not be as suitable for embedded applications compared to a parallel FPGA accelerator. Notably, the 32-core accelerator for the `Grids_30` instance is estimated to consume 1.1W, over two orders of magnitude less than a single cluster CPU. Existing FPGA accelerators[10, 32] for inference on Probabilistic Graphical Models offer great performance, but it isn't clear that they are scalable enough to feasibly solve larger problems[10] or they are optimized for specific applications[32]. The work presented in [10] for example supports a maximum of 29 nodes per Xilinx Virtex 5 LX330 FPGA and typically problems from the UAI competitions[55] sport hundreds if not thousands of variable nodes. The primary contribution here is then a hardware efficient solver, general purpose solver for probabilistic inference that can deduce reasonable solutions to MAP problems in a short amount of time.

8.1 Future Work

Extracting concurrency from SAT is a difficult problem, but the aforementioned contributions show a way forward for future work targeting greater concurrency in SAT. For one, there needs to be a greater focus on tuning SLS solvers for real-world application instances and evaluating their scalability on massively parallel systems. There needs to be a focus on approaches that do not rely on information sharing between cores, in the literature there are many works that illustrate the theoretical and empirical limit of scalability for portfolios that rely on information sharing for SAT[7, 75].

One future direction for the deterministic seeding of SLS threads, is to leverage the covering code concepts from [37] and [28] and combine them with a specialized local search algorithm to accurately bound the number of restarts needed to exhaust the search space. If a distributed and greedy covering code approximation can be devised such that for each thread restart, a code-word representing a unique Hamming ball is generated; the search is translated into the massively parallel task of performing local search within the independent Hamming balls for a solution. By constraining the local search to remain within the Hamming ball, restarting once the local minima within the ball is found, it may be possible to enable massively parallel local search while reducing or eliminating redundant operation across threads.

The hybrid solver approach presented in Chapter 5 frequently reached a point where the CDCL component got stuck solving a sub-problem that was smaller than the overall instance. It is likely that a specially tuned SLS solver and CDCL solver would perform significantly better, especially with regards to heuristics governing the deletion of learned clauses which has been an active area of research recently[100]. Tuning should specifically target instances that are difficult to solve for both SLS and CDCL solvers. Additionally, the parallel hybrid approach could have more benefit on embedded heterogeneous FPGA architectures where a relatively slow CPU core may not be as effective running the CDCL algorithms and could benefit from hardware acceleration. This could be especially useful for self-verification applications such as [130]. Another hybrid approach presented in Chapter 5 created an independent SLS/CDCL hybrid pair per thread, and each thread was climbing its own independent hill. In this approach the incremental CDCL solver acts almost as a tabu that prevents the SLS solver from revisiting the same local minimas. This solver typically behaved in a similar manner to the previous approach where the CDCL solver would get stuck after most of the clauses had been added to the sub-problem. It would be interesting to further develop this architecture, allowing for the oldest local minima clauses to be removed from the CDCL sub-problem. The algorithm would then be incomplete, but it could make for an effective tabu based parallel local search algorithm with a finite tabu length. Similarly, it may be possible to leverage the variable assumption capability of the incremental CDCL solver to reduce the complexity of the CDCL sub-problem based on heuristics generated from the SLS problem.

The WPMS solver presented in Chapter 6 showed potential for performance and scalability even with a very simple search algorithm. This solver would likely benefit from integration of heuristics such as the reduction of hard clause weight presented in the Ramp[49] solver. The parallel solver could also be very effective in replacing Ramp as the local search portion of a hybrid solver such as maxroster[127]. Further work could also be done to tune the probabilistic inference conversion parameters and solver configuration to accelerate these instances past the domain specific solvers and potentially enable inference based decisions in embedded applications.

Finally, it seems like the massive yet affordable FPGA resources available in cloud instances such as Amazon's F1[5] seems like a natural fit for this approach. Implementation and testing of this work on that platform will allow for evaluation of the presented approach with thousands of cores and expanded capacity by leveraging UltraRAM resources available on those parts. The promise of new FPGAs with stacked HBM available would also allow for scaling capacity of the presented hardware solvers without taking a major performance hit in terms of flips/second.

Bibliography

[1] Max-SAT 2016 - Eleventh Max-SAT Evaluation, . URL http://maxsat.ia.udl.cat/introduction/.

[2] MaxSAT Evaluation 2017, . URL http://mse17.cs.helsinki.fi/rankings.html.

[3] SAT Competition 2014 Results, 2014. URL http://satcompetition.org/2014/results.shtml.

[4] The international SAT Competitions web page, 2016. URL http://www.satcompetition.org/.

[5] Amazon EC2 F1 Instances, 2018. URL https://aws.amazon.com/ec2/instance-types/f1/.

[6] Carlos Ansótegui and Joel Gabàs. Solving (Weighted) {Partial MaxSAT} with {ILP}. In *Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2013)*, volume 7874, pages 403–409, 2013. URL https://computational-sustainability.cis.cornell.edu/cpaior2013/pdfs/ansotegui.pdfhttp://www.cis.cornell.edu/ics/cpaior2013/pdfs/ansotegui.pdf.

[7] Alejandro Arbelaez and Philippe Codognet. Massively Parallel Local Search for SAT. *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, pages 57–64, 2012. ISSN 10823409. doi: 10.1109/ICTAI.2012.17. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6495029.

[8] Alejandro Arbelaez and Philippe Codognet. From sequential to parallel local search for SAT. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7832 LNCS, pages 157–168. Springer, Berlin, Heidelberg, 2013. ISBN 9783642371974. doi: 10.1007/978-3-642-37198-1_14. URL http://link.springer.com/10.1007/978-3-642-37198-1{_}14.

[9] Alejandro Arbelaez and Youssef Hamadi. Improving parallel local search for SAT. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6683 LNCS, pages 46–60. Springer, Berlin, Heidelberg, 2011. ISBN 9783642255656. doi: 10.1007/978-3-642-25566-3_4. URL http://link.springer.com/10.1007/978-3-642-25566-3{_}4.

[10] N.B. Asadi, C.W. Fletcher, G Gibeling, J Wawrzynek, W.H. Wong, and G.P. Nolan. ParaLearn: A Massively Parallel, Scalable System for Learning Interaction Networks on FPGAs. *24th International Conference on Supercomputing*, 14:19, 2010. doi: 10.1145/1810085.1810100. URL papers2://publication/uuid/0DE65DC8-8C09-4B03-8DED-FFF83946670D.

[11] Stephen H Bach, Bert Huang, and Lise Getoor. Unifying Local Consistency and MAX SAT Relaxations for Scalable Inference with Rounding Guarantees. *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, 38:46–55, 2015. ISSN 15337928.

[12] Adrian Balint. Engineering Stochastic Local Search for the Satisfiability Problem. *Phd Thesis*, 2013.

[13] Adrian Balint. The probSAT SAT Solver, 2015. URL https://github.com/adrianopolus/probSAT.

[14] Adrian Balint and Andreas Fröhlich. Improving Stochastic Local Search for SAT with a New Probability Distribution. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6175 LNCS, pages 10–15. Springer, Berlin, Heidelberg, 2010. ISBN 3642141854. doi: 10.1007/978-3-642-14186-7_3. URL http://link.springer.com/10.1007/978-3-642-14186-7{_}3.

[15] Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. *Theory and Applications of Satisfiability Testing*, pages 16–29, 2012. URL http://link.springer.com/chapter/10.1007/978-3-642-31612-8{_}3.

[16] Adrian Balint and Uwe Schöning. probsat and pprobsat. *Proceedings of SAT Competition 2014 Solver and Benchmark Descriptions*, 2014:63, 2014.

[17] Tomas Balyo. GitHub - biotomas/ipasir, 2017. URL https://github.com/biotomas/ipasir.

[18] Tomáš Balyo and Carsten Sinz. Parallel Satisfiability. In *Handbook of Parallel Constraint Reasoning*, pages 3–29. Springer International Publishing, Cham, 2018. doi: 10.1007/978-3-319-63516-3_1. URL http://link.springer.com/10.1007/978-3-319-63516-3{_}1.

[19] Tomáš Balyo, Marijn J H Heule, and Matti Järvisalo. SAT COMPETITION 2017 Solver and Benchmark Descriptions. In *Proceedings of SAT COMPETITION 2017*, pages 14–15, 2017. ISBN 9789515100436. URL https://helda.helsinki.fi/

bitstream/handle/10138/224324/sc2017-proceedings.pdf?sequence=4https://helda.helsinki.fi/bitstream/handle/10138/224324/sc2017-proceedings.pdf?sequence=4{%}0Ahttps://helda.helsinki.fi/bitstream/handle/10138/135571/sc2014{_}proceeding.

[20] Sander Beckers, Gorik De Samblanx, Floris De Smedt, Toon Goedemé, Lars Struyf, and Joost Vennekens. Parallel hybrid SAT solving using OpenCL. *Belgian/Netherlands Artificial Intelligence Conference*, 2012. ISSN 15687805.

[21] A. Belov, D. Diepold, M. Heule, and M. Järvisalo. Proceedings of SAT Competition 2014 Solver and Benchmark Descriptions, 2014. URL http://www.satcompetition.org/2014/.

[22] E. Bensana, M. Lemaître, and G. Verfaillie. Earth Observation Satellite Management. *Constraints*, 4(3):293–299, 1999. ISSN 13837133. doi: 10.1023/A:1026488509554.

[23] Jeremias Berg and Matti Jarvisalo. Optimal Correlation Clustering via MaxSAT. In *2013 IEEE 13th International Conference on Data Mining Workshops*, pages 750–757. IEEE, dec 2013. ISBN 978-1-4799-3142-2. doi: 10.1109/ICDMW.2013.99. URL http://ieeexplore.ieee.org/document/6753996/.

[24] Jeremias Berg and Matti Järvisalo. Cost-optimal constrained correlation clustering via weighted partial Maximum Satisfiability. *Artificial Intelligence*, 244:110–142, mar 2017. ISSN 00043702. doi: 10.1016/j.artint.2015.07.001. URL https://www.sciencedirect.com/science/article/pii/S0004370215001022?via{%}3Dihubhttp://linkinghub.elsevier.com/retrieve/pii/S0004370215001022.

[25] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. *FMV Reports Series*, Technical:1–4, 2010.

[26] Armin Biere. Yet another Local Search Solver and Lingeling and Friends Entering the SAT Competition 2014. *SAT COMPETITION 2014 Solver and Benchmark Descriptions*, pages 43–44, 2014. URL http://fmv.jku.at/papers/Biere-SAT-Competition-2014.pdf.

[27] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization. *ACM Computing Surveys*, 35(3):268–308, sep 2003. ISSN 03600300. doi: 10.1145/937503.937505. URL http://portal.acm.org/citation.cfm?doid=937503.937505.

[28] Tobias Brueggemann and Walter Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1-3):303–313, dec 2004. ISSN 0304-3975. doi: 10.1016/J.TCS.2004.08.002. URL https://www.sciencedirect.com/science/article/pii/S0304397504005146.

[29] Shaowei Cai, Chuan Luo, John Thornton, and Kaile Su. Tailoring Local Search for Partial {MaxSAT}. *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 2623–2629, 2014. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8141.

[30] Shaowei Cai, Chuan Luo, and Haochen Zhang. From Decimation to Local Search and Back: A New Approach to MaxSAT. pages 571–577, 2017. ISSN 10450823.

[31] Yibin Chen, Sean Safarpour, Joao Marques-Silva, and Andreas Veneris. Automated design debugging with maximum satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(11):1804–1817, nov 2010. ISSN 02780070. doi: 10.1109/TCAD.2010.2061270. URL http://ieeexplore.ieee.org/document/5605301/.

[32] Jungwook Choi and Rob A. Rutenbar. Hardware implementation of MRF map inference on an FPGA platform. *Proceedings - 22nd International Conference on*

Field Programmable Logic and Applications, FPL 2012, pages 209–216, 2012. doi: 10.1109/FPL.2012.6339183.

[33] Philippe Codognet, Danny Munera, Daniel Diaz, and Salvador Abreu. Parallel Local Search. In *Handbook of Parallel Constraint Reasoning*, pages 381–417. Springer International Publishing, Cham, 2018. doi: 10.1007/978-3-319-63516-3_10. URL http://link.springer.com/10.1007/978-3-319-63516-3{_}10.

[34] David A Cohen, Martin C Cooper, and Peter G Jeavons. A Complete Characterization of Complexity for Boolean Constraint Optimization Problems. *Cp*, pages 212–226, 2004. ISSN 03029743 16113349.

[35] Stephen A. Cook. The complexity of theorem-proving procedures, 1971. ISSN 08985626. URL http://portal.acm.org/citation.cfm?doid=800157.805047.

[36] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. CUD@SAT: SAT solving on GPUs. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(3):293–316, may 2015. ISSN 0952-813X. doi: 10.1080/0952813X.2014.954274. URL http://www.tandfonline.com/doi/full/10.1080/0952813X.2014.954274.

[37] Evgeny Dantsin, Andreas Goerdt, Edward A Hirsch, Ravi Kannan, Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic (2−2/(k+1))n algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, oct 2002. ISSN 0304-3975. doi: 10.1016/S0304-3975(01)00174-8. URL https://www.sciencedirect.com/science/article/pii/S0304397501001748.

[38] Adnan Darwiche and Knot Pipatsrisawat. Complete algorithms. *Handbook of Satisfiability*, 185:99–130, 2009.

[39] Adnan Darwiche and Knot Pipsatrisawat. Complete Algorithms. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 3, pages 99–126. IOS Press, Amsterdam, 2009. ISBN 978-1-58603-929-5.

[40] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7962 LNCS, pages 166–181. Springer, Berlin, Heidelberg, 2013. ISBN 9783642390708. doi: 10.1007/978-3-642-39071-5_13. URL http://link.springer.com/10.1007/978-3-642-39071-5{_}13.

[41] John D. Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. A practical reconfigurable hardware accelerator for Boolean satisfiability solvers. In *Proceedings of the 45th annual conference on Design automation - DAC '08*, page 780, New York, New York, USA, jun 2008. ACM Press. ISBN 9781605581156. doi: 10.1145/1391469.1391669. URL http://dl.acm.org/citation.cfm?id=1391469.1391669.

[42] JohnD. Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. Designing an Efficient Hardware Implication Accelerator for SAT Solving. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*, pages 48–62. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-79718-0. doi: 10.1007/978-3-540-79719-7_6. URL http://dx.doi.org/10.1007/978-3-540-79719-7_6.

[43] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, jul 1962. ISSN 00010782. URL http://dl.acm.org/citation.cfm?id=368273.368557.

[44] J.T. de Sousa, J. M. Silva, and M. Abramovici. A Configware/Software Approach

to SAT Solving. mar 2001. URL https://www.researchgate.net/publication/2414583{_}A{_}ConfigwareSoftware{_}Approach{_}to{_}SAT{_}Solving.

[45] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. *Theory and Applications of Satisfiability Testing*, pages 61–75, 2005. ISSN 03029743. doi: 10.1007/11499107{_}5. URL http://link.springer.com/chapter/10.1007/11499107{_}5.

[46] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, pages 502–518, 2004. ISSN 03029743. doi: 10.1007/978-3-540-24605-3_37. URL http://link.springer.com/chapter/10.1007/978-3-540-24605-3{_}37.

[47] Niklas Een, Alan Mishchenko, and S Niklas. Applying Logic Synthesis for Speeding Up SAT. *Sat*, pages 1–14, 2007. ISSN 03029743. doi: 10.1007/978-3-540-72788-0{_}26.

[48] Paul Eggert. glibc/random_r.c at master · lattera/glibc · GitHub, 2012. URL https://github.com/lattera/glibc/blob/master/stdlib/random{_}r.c.

[49] Yi Fan, Zongjie Ma, Kaile Su, and Abdul Sattar. Ramp : A Local Search Solver based on Make-positive Variables. pages 4321–4322, 2016. URL http://maxsat.ia.udl.cat/solvers/11/Ramp-201604151325.

[50] Michael S Fang, Lei and Hsiao. Boosting sat solver performance via a new hybrid approach. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:243–261, 2008. ISSN 0305-7410. doi: 10.1017/S0305741000035475. URL https://www.satassociation.org/jsat/index.php/jsat/article/view/65.

[51] Andreas Frohlich, Armin Biere, Christoph M Wintersteiger, and Youssef Hamadi.

Stochastic Local Search for Satisfiability Modulo Theories. *Aaai-2015*, pages 1136–1143, 2015.

[52] Hironori Fujii and Noriyuki Fujimoto. GPU Acceleration of BCP Procedure for SAT Algorithms. *Elrond.Informatik.Tu-Freiberg.De*, 2012. URL http://elrond.informatik.tu-freiberg.de/papers/WorldComp2012/PDP6166.pdf{%}5Cnhttp://world-comp.org/p2012/PDP6166.pdf.

[53] Johan De Gelas. Single Core Integer Performance With SPEC CPU2006 - The Intel Xeon E5 v4 Review: Testing Broadwell-EP With Demanding Server Workloads. URL https://www.anandtech.com/show/10158/the-intel-xeon-e5-v4-review/8.

[54] Patrice Godefroid, Michael Y. Levin, and David a. Molnar. Automated Whitebox Fuzz Testing. *Search*, 9(July):pdf, 2008. ISSN 1064-3745. doi: 10.1007/978-3-642-02652-2_1. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.151.9430{&}rep=rep1{&}type=pdf.

[55] Vibhav Gogate. UAI 2014 Inference Competition, 2014. URL http://www.hlt.utdallas.edu/{~}vgogate/uai14-competition/index.html.

[56] Thomas L. Griffiths and Alan Yuille. A primer on probabilistic inference. In *The Probabilistic Mind: Prospects for Bayesian cognitive science*. 2012. ISBN 9780191695971. doi: 10.1093/acprof:oso/9780199216093.003.0002.

[57] K. Gulati, M. Waghmode, S.P. Khatri, and W. Shi. Efficient, scalable hardware engine for Boolean satisfiability and unsatisfiable core extraction. *IET Computers & Digital Techniques*, 2(3):214, may 2008. ISSN 17518601. doi: 10.1049/iet-cdt:20060221. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4492953.

[58] Kanupriya Gulati, Suganth Paul, Sunil P. Khatri, Srinivas Patil, and Abhijit Jas.

FPGA-based hardware acceleration for Boolean satisfiability. *ACM Transactions on Design Automation of Electronic Systems*, 14(2):1–11, mar 2009. ISSN 10844309. doi: 10.1145/1497561.1497576. URL http://dl.acm.org/citation.cfm?id=1497561.1497576.

[59] Leopold Haller and Satnam Singh. Relieving capacity limits on FPGA-based SAT-solvers. *Formal Methods in Computer Aided Design*, pages 217–220, 2010.

[60] Youssef Hamadi and Christoph Wintersteiger. Seven Challenges in Parallel SAT Solving. *AI Magazine*, 34(2):99, jun 2013. ISSN 0738-4602. doi: 10.1609/aimag.v34i2.2450. URL https://aaai.org/ojs/index.php/aimagazine/article/view/2450.

[61] Youssef Hamadi and Christoph M Wintersteiger. Seven Challenges in Parallel SAT Solving. *AAAI Conference on Artificial Intelligence*, pages 2120–2125, 2012. ISSN 07384602. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5001.

[62] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, (6):245–262, nov 2009. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.6667{&}rep=rep1{&}type=pdf.

[63] Marijn J H Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7261 LNCS:50–65, 2012. ISSN 03029743. doi: 10.1007/978-3-642-34188-5_8.

[64] Marijn J. H. Heule, Oliver Kullmann, and Armin Biere. Cube-and-Conquer for Satisfiability. In *Handbook of Parallel Constraint Reasoning*, chapter 2, pages 31–59.

Springer International Publishing, Cham, 2018. doi: 10.1007/978-3-319-63516-3_2. URL http://link.springer.com/10.1007/978-3-319-63516-3{_}2.

[65] Steffen Holldobler, Norbert Manthey, Van Hau Nguyen, Julian Stecklina, and Peter Steinke. A short overview on modern parallel SAT-solvers. *2011 International Conference on Advanced Computer Science and Information Systems*, (i):201–206, 2011.

[66] Holger H Hoos. On the Run-time Behaviour of Stochastic Local Search Algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, {USA.}*, pages 661–666, 1999. ISBN 0262511061. URL https://pdfs.semanticscholar.org/370c/f8b55191fdc295c1aa871ad199ec269f7b4a.pdfhttp://www.aaai.org/Library/AAAI/1999/aaai99-094.php.

[67] Holger H. Hoos. SATLIB - Benchmark Problems, 2011. URL http://www.cs.ubc.ca/{~}hoos/SATLIB/benchm.html.

[68] Bert Huang. berthuang / weighted_max_sat — Bitbucket, 2018. URL https://bitbucket.org/berthuang/weighted{_}max{_}sat/src/master/.

[69] Teodor Ivan and El Mostapha Aboulhamid. An Efficient Hardware Implementation of a SAT Problem Solver on FPGA. In *2013 Euromicro Conference on Digital System Design*, pages 209–216. IEEE, sep 2013. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6628279.

[70] Farah Juma, Eric I. Hsu, and Sheila A. McIlraith. Preference-based planning via MaxSAT. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7310 LNAI:109–120, 2012. ISSN 03029743. doi: 10.1007/978-3-642-30353-1_10.

[71] K. Kanazawa and T. Maruyama. An FPGA solver for WSAT algorithms. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 83–88. IEEE, 2005. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1515703.

[72] Kenji Kanazawa and Tsutomu Maruyama. An Approach for Solving Large SAT Problems on FPGA. *ACM Transactions on Reconfigurable on Technology and Systems*, 4 (1), 2010. doi: 10.1145/1857927.1857937. URL http://dl.acm.org/citation.cfm?id=1857937.

[73] Kenji Kanazawa and Tsutomu Maruyama. FPGA acceleration of SAT/Max-SAT solving using variable-way cache. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, sep 2014. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6927405.

[74] Kenji Kanazawa and Tsutomu Maruyama. An Approach for Solving SAT/MaxSAT-Encoded Formal Verification Problems on FPGA. (8), 2017. doi: 10.1587/transinf.2016EDP7487. URL https://www.jstage.jst.go.jp/article/transinf/E100.D/8/E100.D{_}2016EDP7487/{_}pdf/-char/ja.

[75] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and Parallelizability Barriers to the Efficient Parallelization of SAT Solvers.pdf. *Aaai*, 2013.

[76] Ashiqur R. KhudaBukhsh, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. *Artificial Intelligence*, 232:20–42, 2016. ISSN 00043702. doi: 10.1016/j.artint.2015.11.002. URL http://www.cs.ubc.ca/labs/beta/Projects/SATenstein/2016-AIJ-SATenstein.pdf.

[77] Lukas Kroc, Ashish Sabharwal, Carla P. Gomes, and Bart Selman. Integrating systematic and local search paradigms: A new strategy for MaxSAT. *IJCAI International Joint Conference on Artificial Intelligence*, pages 544–551, 2009. ISSN 10450823.

[78] Chu Min Li and Felip Manyà. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*, chapter 19, pages 613–966. IOS Press, Amsterdam, 2009.

[79] Chumin Li, Chong Huang, and Ruchu Xu. Balance between intensification and diversification: two sides of the same coin. In *Proc. of SAT Competition 2013*, pages 10–11, 2013. URL https://helda.helsinki.fi/bitstream/handle/10138/40026/sc2013{_}proceedings.pdf?sequence=2.

[80] Sixue Liu, Yulong Ceng, and Gerard de Melo. A Probability Distribution Strategy with Efficient Clause Selection for Hard Max-SAT Formulas. pages 1–11, 2016. URL http://arxiv.org/abs/1610.00442.

[81] Chuan Luo, Shaowei Cai, Wei Wu, Zhong Jie, and Kaile Su. CCLS: An Efficient Local Search Algorithm for Weighted Maximum Satisfiability. *IEEE Transactions on Computers*, 64(7):1830–1843, jul 2015. ISSN 00189340. doi: 10.1109/TC.2014.2346196. URL http://ieeexplore.ieee.org/document/6874523/.

[82] Chuan Luo, Shaowei Cai, Kaile Su, and Wenxuan Huang. CCEHC: An efficient local search algorithm for weighted partial maximum satisfiability. *Artificial Intelligence*, 243(C):26–44, feb 2017. ISSN 00043702. doi: 10.1016/j.artint.2016.11.001. URL http://linkinghub.elsevier.com/retrieve/pii/S0004370216301382.

[83] Inês Lynce, Vasco Manquinho, and Ruben Martins. Parallel Maximum Satisfiability. In *Handbook of Parallel Constraint Reasoning*, pages 61–99. Springer International Publishing, Cham, 2018. doi: 10.1007/978-3-319-63516-3_3. URL http://link.springer.com/10.1007/978-3-319-63516-3{_}3.

[84] Ravi Mangal, Xin Zhang, Aditya Kamath, Aditya V. Nori, and Mayur Naik. Scaling relational inference using proofs and refutations. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, (Papadimitriou 1994):3278–3286, 2016.

[85] Joao Marques-Silva. Practical applications of Boolean Satisfiability. In *2008 9th International Workshop on Discrete Event Systems*, pages 74–80. IEEE, 2008. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4605925.

[86] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 185, chapter 4, pages 131–153. Amsterdam, 2009. ISBN 9781586039295. doi: 10.3233/978-1-58603-929-5-131.

[87] J.P. Marques-Silva and K.A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, may 1999. ISSN 00189340. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=769433.

[88] Ruben Martins. Solving RNA Alignment with MaxSAT. *MaxSAT Evaluation 2017 Solver and Benchmark Descriptions*, pages 29–30, 2017.

[89] Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, jul 2012. ISSN 1383-7133. doi: 10.1007/s10601-012-9121-3. URL http://link.springer.com/10.1007/s10601-012-9121-3.

[90] Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A Modular MaxSAT Solver,. pages 438–445. Springer, Cham, 2014. doi: 10.1007/978-3-319-09284-3_33. URL http://link.springer.com/10.1007/978-3-319-09284-3{_}33.

[91] Ruben Martins, Vasco Manquinho, and Inês Lynce. Deterministic Parallel MaxSAT

Solving. *International Journal on Artificial Intelligence Tools*, 24(03):1550005, jun 2015. ISSN 0218-2130. doi: 10.1142/s0218213015500050. URL http://www.worldscientific.com/doi/abs/10.1142/s0218213015500050.

[92] Rúben Carlos Gonçalves Martins. *Parallel Search for Maximum Satisfiability*. PhD thesis, UNIVERSIDADE DE LISBOA INSTITUTO SUPERIOR TECNICO, 2013. URL http://sat.inesc-id.pt/{~}ruben/papers/martins-phd.pdf.

[93] Math6068. GitHub - math6068/Ramp. URL https://github.com/math6068/Ramp.

[94] Q Meyer, F Schönfeld, M Stamminger, and R Wanka. 3-SAT on CUDA: Towards a massively parallel SAT solver. In *2010 International Conference on High Performance Computing & Simulation*, pages 306–313. IEEE, jun 2010. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5547116.

[95] Botond Molnár, Melinda Varga, Zoltán Toroczkai, and Mária Ercsey-Ravasz. A high-performance analog Max-SAT solver and its application to Ramsey numbers. 2018. URL https://arxiv.org/pdf/1801.06620.pdf.

[96] M W Moskewicz, C F Madigan, Ying Zhao, Lintao Zhang, and S Malik. Chaff: Engineering an Efficient SAT Solver. *Design Automation Conference*, pages 530–535, 2001. ISSN 0738-100X. doi: 10.1109/DAC.2001.156196.

[97] Miguel Neves, Inês Lynce, and Vasco Manquinho. DistMS: A Non-Portfolio Distributed Solver for Maximum Satisfiability. may 2015. URL http://arxiv.org/abs/1505.02408.

[98] Aina Niemetz, Mathias Preiner, and Armin Biere. *Precise and Complete Propagation Based Local Search for Satisfiability Modulo Theories*, pages 199–217. Springer International Publishing, Cham, 2016. ISBN 978-3-319-41528-

4. doi: 10.1007/978-3-319-41528-4_11. URL http://dx.doi.org/10.1007/978-3-319-41528-4{_}11.

[99] Melissa E O 'neill. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random. Technical report, Harvey Mudd College, Claremont, CA, 2014. URL https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdfwww.cs.hmc.edu.

[100] Chanseok Oh. Between SAT and UNSAT: The Fundamental Difference in CDCL SAT. In Sean Heule, Marijn and Weaver, editor, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 307–323. Springer International Publishing, 2015. ISBN 978-3-642-02776-5. doi: 10.1007/978-3-319-24318-4{_}23. URL http://hdl.handle.net/10512/99999190http://link.springer.com/10.1007/978-3-319-24318-4{_}23.

[101] Melissa E. O'Neill. PCG, A Family of Better Random Number Generators | PCG, A Better Random Number Generator, 2018. URL http://www.pcg-random.org/.

[102] Lars Otten. GitHub - lotten/daoopt: Sequential and distributed AND/OR Branch and Bound for MPE (max-product) problems over graphical models like Bayes and Markov networks., 2015. URL https://github.com/lotten/daoopt.

[103] Lars Otten. AND / OR Branch-and-Bound on a Computational Grid. *Journal of Artificial Intelligence Research*, 59:351–435, 2017.

[104] Lars Otten and Rina Dechter. Parallelizing AND / OR Branch-and-Bound. pages 1–80.

[105] James D Park. Using weighted max-SAT engines to solve MPE. In *Proc.\ of AAAI*, pages 682–687, 2002.

[106] Marco Platzner and Giovanni De Micheli. Acceleration of satisfiability algorithms by reconfigurable hardware. In ReinerW. Hartenstein and Andres Keevallik, editors, *Field-Programmable Logic and Applications From FPGAs to Computing Paradigm*, volume 1482 of *Lecture Notes in Computer Science*, pages 69–78. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-64948-9. doi: 10.1007/BFb0055234. URL http://dx.doi.org/10.1007/BFb0055234.

[107] D.K. Pradhan, D. Kagaris, and R. Gambhir. A Hamming distance based test pattern generator with improved fault coverage. In *11th IEEE International On-Line Testing Symposium*, pages 221–226. IEEE, 2005. ISBN 0-7695-2406-0. doi: 10.1109/IOLTS.2005.6. URL http://ieeexplore.ieee.org/document/1498165/.

[108] Steven David Prestwich. Cnf encodings. *Handbook of satisfiability*, 185:75–97, 2009.

[109] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. SC13 Tutorial: Hybrid MPI and OpenMP Parallel Programming, 2013. ISSN 1066-6192. URL https://www.openmp.org/press-release/sc13-tutorial-hybrid-mpi-openmp-parallel-programming/.

[110] Phillip E. Ross. Why CPU Frequency Stalled - IEEE Spectrum, 2008. URL https://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled.

[111] M Safar, M W El-Kharashi, M Shalan, and a Salem. A reconfigurable, pipelined, conflict directed jumping search SAT solver. *2011 Design, Automation & Test in Europe*, pages 1–6, 2011. ISSN 15301591. doi: 10.1109/DATE.2011.5763199. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5763199.

[112] Mona Safar, M. Watheq El-Kharashi, and Ashraf Salem. FPGA-based SAT solver. In *Canadian Conference on Electrical and Computer Engineering*, number May, pages 1901–1904, 2006. ISBN 1424400384. doi: 10.1109/CCECE.2006.277452.

[113] Mona Safar, Mohamed Shalan, M. Watheq El-Kharashi, and Ashraf Salem. A Shift Register based Clause Evaluator for Reconfigurable SAT Solver. *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6, 2007. doi: 10.1109/DATE.2007.364583. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4211788.

[114] Paul Saikko, Jeremias Berg, and Matti Järvisalo. LMHS: A SAT-IP hybrid MaxSAT solver. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9710, pages 539–546. Springer, Cham, 2016. ISBN 9783319409696. doi: 10.1007/978-3-319-40970-2_34. URL http://link.springer.com/10.1007/978-3-319-40970-2{_}34.

[115] Shohei Sassa, Kenji Kanazawa, Shaowei Cai, and Moritoshi Yasunaga. An FPGA Solver for Partial MaxSAT Problems Based on Stochastic Local Search. *SIGARCH Comput. Archit. News*, 44(4):32–37, 2017. ISSN 0163-5964. doi: 10.1145/3039902.3039909. URL http://delivery.acm.org/10.1145/3040000/3039909/p32-sassa.pdf?ip=128.173.52.3{&}id=3039909{&}acc=ACTIVESERVICE{&}key=B33240AC40EC9E30.80AE0C8B3B97B250.4D4702B0C3E38B35.4D4702B0C3E38B35{&}CFID=846658608{&}CFTOKEN=80249666{&}{_}{_}acm{_}{_}=1514768705{_}43031086487fd5f3a39320cf.

[116] T. Schoning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*,

pages 410–414. IEEE Comput. Soc. ISBN 0-7695-0409-4. doi: 10.1109/SFFCS.1999.814612. URL http://ieeexplore.ieee.org/document/814612/.

[117] Peter Selinger. The GLIBC pseudo-random number generator, 2007. URL https://www.mathstat.dal.ca/{~}selinger/random/.

[118] B Selman, H Kautz, and B Cohen. Noise strategies for local search. *AAAI/IAAI Proceedings*, (1990):337–343, 1998.

[119] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, volume 20, pages 440–446, 1992. ISBN 0-262-51063-4.

[120] Bart Selman, Bart Selman, Henry Kautz, and Bram Cohen. Local Search Strategies for Satisfiability Testing. *DIMACS SERIES IN DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE*, 26:521—-532, 1995. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.2207.

[121] Laurent Simon. The Glucose SAT Solver, 2016. URL http://www.labri.fr/perso/lsimon/glucose/.

[122] I. Skliarova and A. de Brito Ferrari. Reconfigurable hardware SAT solvers: a survey of systems. *IEEE Transactions on Computers*, 53(11):1449–1461, nov 2004. ISSN 0018-9340. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1336765.

[123] I Skliarova and A B Ferrari. A software/reconfigurable hardware {SAT} solver. *IEEE Transactions on Very Large Scale Intergration (VLSI) Systems*, 12(4):408–419, 2004. ISSN 1063-8210. doi: 10.1109/TVLSI.2004.825859.

[124] Ali Asgar Sohanghpurwala and Peter Athanas. An effective probability distribution SAT solver on reconfigurable hardware. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, nov 2016. ISBN 978-1-5090-3707-0. doi: 10.1109/ReConFig.2016.7857150. URL `http://ieeexplore.ieee.org/document/7857150/`.

[125] Ali Asgar Sohanghpurwala, Mohamed W. Hassan, and Peter Athanas. Hardware accelerated SAT solvers—A survey. *Journal of Parallel and Distributed Computing*, 2016. ISSN 07437315. doi: 10.1016/j.jpdc.2016.12.014. URL `http://www.sciencedirect.com/science/article/pii/S0743731516301903`.

[126] Ivor Spence. Generator of benchmarks for the satisfiability problem (sgen), 2009. URL `http://www.cs.qub.ac.uk/{~}i.spence/sgen/`.

[127] Takayuki Sugawara. MaxRoster:Solver Description. In Carlos Ansotegui, Fahiem Bacchus, Matti Järvisalo, and Ruben Martins, editors, *MaxSAT Evaluation 2017 Solver and Benchmark Descriptions*, page 12, 2017.

[128] Jason Thong. *FPGA Acceleration of Decision-Based Problems using Heterogeneous Computing*. PhD thesis, McMaster University, 2014.

[129] Jason Thong and Nicola Nicolici. FPGA acceleration of enhanced boolean constraint propagation for SAT solvers. *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, pages 234–241, 2013. ISSN 10923152. doi: 10.1109/ICCAD.2013.6691124.

[130] Buse Ustaoglu, Sebastian Huhn, Daniel Große, and Rolf Drechsler. SAT-Lancer. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI - GLSVLSI '18*, pages 479–482, New York, New York, USA, 2018. ACM Press. ISBN 9781450357241. doi:

10.1145/3194554.3194643. URL http://dl.acm.org/citation.cfm?doid=3194554.3194643.

[131] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Day. Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance (WP485). 2017. URL https://www.xilinx.com/support/documentation/white{_}papers/wp485-hbm.pdf.

[132] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32: 565–606, 2008. ISSN 10769757. doi: 10.1613/jair.2490.

[133] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2-3):107–143, feb 2007. ISSN 00043702. doi: 10.1016/j.artint.2006.11.005. URL https://www.sciencedirect.com/science/article/pii/S0004370206001408http://linkinghub.elsevier.com/retrieve/pii/S0004370206001408.

[134] Zhongda Yuan, Yuchun Ma, and Jinian Bian. SMPP: Generic SAT solver over reconfigurable hardware accelerator. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012*, pages 443–448, 2012. doi: 10.1109/IPDPSW.2012.57.

[135] Lei Zhang and Fahiem Bacchus. MAXSAT Heuristics for Cost Optimal Planning. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 1846–1852, 2012. ISBN 9781577355687. doi: 10.1.1.363.6339.

[136] Peixin Zhong, M. Martonosi, P. Ashar, and S. Malik. Accelerating Boolean satisfiability with configurable hardware. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, pages 186–195. IEEE

Comput. Soc, 1998. ISBN 0-8186-8900-5. doi: 10.1109/FPGA.1998.707896. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=707896.

[137] Peixin Zhong, M. Martonosi, P. Ashar, and S. Malik. Using configurable computing to accelerate Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):861–868, jun 1999. ISSN 02780070. doi: 10.1109/43.766733. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=766733.

[138] Bartosz Zoltak. Research Project - ”VMPC One-Way Function - is P=NP?”, 2016. URL http://www.vmpcfunction.com/pnp.php.