JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

An Evaluation of Bit-Parallelization applied to Failed Literal Probing

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Informatik

Eingereicht von:
Robert Aistleitner, BSc.

Angefertigt am:
Institute for Formal Models and Verification

Beurteilung:
Univ.-Prof. Dr. Armin Biere

Linz, Februar, 2013

Abstract

The satisfiability problem (SAT) can be used to encode many problems of other domains since it is the first one to be proven NP-complete. Powerful SAT solvers have been developed in the last years which can handle large real-world problems in a reasonable amount of time. Today's significant challenges on SAT solvers include the transition to parallel computer architectures. Various techniques have been developed to utilize multiple cores, but those are limited to multi-core CPU systems.

This thesis evaluates another concept of parallelism than what is state-of-the-art today. An algorithm is developed that is based on a 'streaming' principle, like it is optimal for massively parallel systems, including modern graphics cards. To take advantage of this principle, major changes to existing algorithms are necessary, first of all to avoid branches whenever possible. This also gives the opportunity to add parallelism in the sense of bit-parallelism.

As the evaluation will show, there is a good chance that the implemented algorithm will perform well on massively parallel systems. Even on regular computer systems further optimization potential is conceivable.

Zusammenfassung

Das “Satisfiability Problem (SAT)” bietet die Möglichkeit, auch viele Probleme aus anderen Problemklassen abzubilden. Der Grund dafür liegt darin, dass es das erste Problem war, das nachweislich der Klasse “NP-vollständig” zugeordnet werden konnte. In den letzten Jahren konnten leistungsstarke Algorithmen zur Lösung dieser Probleme (“SAT solver”) entwickelt werden, welche für viele reelle Aufgaben eingesetzt werden können. Neue Entwicklungen im Bereich von Multiprozessoren stellen die Forschung heutzutage vor neue Herausforderungen. Neue, parallele Algorithmen wurden entwickelt, um die neu gewonnenen Ressourcen ausnützen zu können, jedoch sind diese meist auf Mehrkern-Prozessoren ausgelegt.

Diese Arbeit versucht die Parallelisierung auf einem alternativen Weg anzugehen, entgegen dem aktuellen Stand der Technik. Zu diesem Zweck wird ein Algorithmus entwickelt, der nach einem “streaming”-Konzept arbeitet und so besonders für massiv-parallele Systeme wie Grafikprozessoren gut geeignet sein soll. Um diesen Algorithmus umzusetzen, sind grundlegende Änderungen im Kern üblicher Algorithmen notwendig. Diese Änderungen ermöglichen außerdem den Einsatz von Bit-Parallelismus als zweite Parallelisierungstechnik.

Wie in der Evaluierungsphase der Arbeit zu sehen sein wird, hat der entwickelte Algorithmus durchaus Potential um auf massiv-parallelen Systemen gute Ergebnisse zu erzielen. Die Arbeit zeigt außerdem noch andere Optimierungsmöglichkeiten für die Grundzüge des Algorithmus auf.

Danksagung

Zu allererst möchte ich meinem Betreuer, Prof. Armin Biere, auf das Beste danken. Ich bin überwältigt, wieviel Aufmerksamkeit und Hilfsbereitschaft ich in der Zeit meiner Masterarbeit von Ihm erhalten durfte. Eine große Hilfe war sein enormes Fachwissen, welches er jederzeit gerne mit mir geteilt hat. Viele konstruktive Besprechungen und ermunternde Worte haben mich schließlich an das lang ersehnte Ziel geführt.

Der größte Dank gebührt meinen Eltern, Elfriede und Josef. Ich habe es lange Zeit für selbstverständlich gehalten, die volle Freiheit über meinen Weg im Leben zu haben. Wie sich aber herausstellte, haben nicht alle dasselbe Glück und müssen in vielen Bereichen des Lebens Abstriche machen.

Ich bin unendlich dankbar, dass ihr mich in jedem meiner Schritte ermutigt und bekräftigt, vor allem aber, dass Ihr es überhaupt ermöglicht habt. Was ich euch nun zurückgeben kann ist meine große Dankbarkeit. Ich erfülle euch hoffentlich mit Stolz.

Ein weiterer Dank gebührt meiner Familie sowie meinen Freunden, die immer ein offenes Ohr für mich hatten - sei es aus Freude oder Kummer. Es hilft ungemein jemanden zu haben, der einen auf andere Gedanken bringt, um sich später wieder neu fokussieren zu können. Danke auch für das Interesse an meiner Masterarbeit, auch wenn es oft schwierig war und ist, den Kern der Sache für Laien verständlich aufzubereiten.

Zu guter Letzt will ich meiner Freundin Monika für ihre Unterstützung danken. Sie hat mich regelmäßig dazu motiviert und mir die nötige Zeit gelassen, sowohl mein Studium als auch diese Arbeit zu vollenden. Ich hoffe ihr im Laufe unseres gemeinsamen Lebens viel davon zurückgeben zu können.

Contents

1 Introduction

There exist many problems that can be encoded as propositional satisfiability (SAT) problems. The fields of applications are diverse, including formal verification of hardware and software design ([12, 33]). There even exist compilers that generate SAT problems automatically from a higher abstraction level (e.g. [8]). There are hundreds of applications for SAT solving, for more examples there exists a really comprehensive survey by Gu et al. [18]. A historically interesting fact is that SAT problems were the first to be classified as NP-complete by Cook in 1971 [10], hence theoretically all other problems in NP can be translated into a SAT problem (e.g. quasigroup problems [40]).

As already mentioned, deciding satisfiability is NP-complete and therefore really hard to implement efficiently. In theory you have to check every possible assignment of each variable if it satisfies the given propositional formula, which leads to 2^n checks. It is totally clear that there is no effective way to find a solution for large problems with such a naive approach.

Martin Davis and Hilary Putnam made the first step in a more sophisticated way of computing the satisfiability of a problem in 1960 [14]. The problem at that time was the very limited computational power - especially in terms of limited main memory. Only very small instances were solvable with the available systems, also because algorithms were very memory-intense. Nevertheless many improvements have been made to this very basic algorithm until now - state-of-the-art SAT solvers still use the basic principles of Davis and Putnam.

SAT solvers nowadays can handle real-world problems not only by improved computational power, but also because of extensive research results in the past 20 years. New algorithms have been developed that utilize powerful techniques like machine learning or probability theory applied to new developed heuristics.

Another significant issue today concerns multiprocessing and in general parallelism, this is a very recent topic in SAT solving research. There exist different approaches of utilizing new multiprocessor architectures, which can be found in Sect. 3. This work deals with a rather unconventional type of parallelism which is discussed at the beginning of Sect. 1.1.

This thesis will give a general overview of SAT solving starting with a history including the basic types of SAT solvers in Sect. 2. It also introduces current techniques of parallel SAT solving in the subsequent Sect. 3. Failed literal probing - the algorithm that has been implemented in this thesis - is one kind of probing-based simplification techniques, which are described in Sect. 4. Related work regarding bit-parallel SAT algorithms is discussed in Sect. 5. Sections 6 and 7 show details of how the algorithm is implemented followed by the experiments section including benchmarks and their evaluation. The last Sect. 8 subsumes the results of the thesis and gives a short outlook on future work to improve the developed algorithm.

1.1 Motivation

The motivation of this work is to go a different way of implementing a parallel SAT solving algorithm. Research concentrates on enhancing heuristics and the usage of high-level parallelism. These algorithms are heavily based on random memory accesses and many complex procedures. This is acceptable for current CPU architectures although caches are generally not utilized well. Other processor architectures like graphics processing units (GPUs) do not allow such a programming paradigm (at least they do not work efficiently with them).

Different from a CPU which works kind of control-flow based, a GPU works with data-flows. The difference is that a GPU works massively parallel and executes the same action (e.g. pixel transformation) on distinct data (pixels). Originally these operations were very simple or even fixed. By introducing modern real-time computer graphics it was necessary to handle more complex operations (so-called shaders). These small programs were the preliminary stage of today's 'General Purpose GPUs' (GPGPUs).

These GPGPUs made it possible to use the processing power of the highly parallel GPU for many other non-graphic applications. The task of porting an existing algorithm to work efficiently on a graphics card still is a very challenging task.

The most challenging one when developing GPGPU algorithms is to avoid branch divergence. A graphics card executes blocks of threads exactly in parallel (exactly means that the same instructions are executed on different data at the same time; also called SIMD - *single instruction multiple data*). If these instructions contain branches, different data can lead to different paths, which slows parallel execution down. This effect occurs because threads have to execute different code which violates the need to have same instructions for all threads. We will see later in this thesis how this influences the development of the targeted algorithm.

Contrary to what the reader might expect, this work does not result in a GPGPU algorithm, but should be an evaluation and a first step into that direction. This idea of a 'streaming' algorithm is also applicable for CPUs because caches may be better utilized and result in good performance. The development of a branch-less algorithm as well as the enhancement to utilize bit-parallelism itself is a challenging task. Since a fully functional SAT solving algorithm is very stochastic in its nature, a more deterministic and simple algorithm has been chosen for this work: failed literal probing. The core functionality which takes most of the execution time is equal for both algorithms.

This work should result in an algorithm that can be ported to a GPGPU implementation without having much troubles related to branch divergence. Another aim is to evaluate if the use of bit-parallelism can obtain good results compared to a basic implementation. Other optimizations should also be evaluated for their effectiveness (e.g. multi-threading, cache optimizations, SSE operations).

1.2 Contribution

This thesis answers the following questions:

- Is it possible to develop a failed literal probing algorithm that behaves in a data streaming manner?

- Can the use of bit-parallelism speed-up this algorithm and to which extent?
- Does this approach have potential to perform well on massively parallel systems?
- Which optimizations are conceivable regarding prospects of a modern computing system?
- What are the possible drawbacks of the targeted algorithm?
- Are there potential optimizations and additional functionality which are not part of this work?

2 SAT Solving History

When speaking of satisfying a CNF formula it means to find a set of variable assignments so that the whole formula is satisfied (resolves to true). SAT solving algorithms can be split in two top-level categories: incomplete and complete SAT solvers. Both of them are well researched and are recent topics but differ in a major property: incomplete SAT solvers only try to prove satisfiability whereas complete SAT solvers are able to prove satisfiability as well as unsatisfiability. This work concentrates on complete SAT algorithms that originate from the Davis-Putnam procedure but a short insight in incomplete SAT is given nevertheless.

2.1 Preliminaries

SAT problems are generally encoded as propositional formulas in conjunctive normal form (CNF). The only symbols used in a formula are $\wedge$ for conjunction, $\vee$ for disjunction and $\neg$ for negation (which is only allowed directly in front of literals). The following definitions show how a CNF formula is built and which properties it can have.

Definition 1. *A CNF formula is a conjunction of clauses $\bigwedge_i c_i$. A CNF formula is satisfied if all containing clauses c_i are satisfied.*

Definition 2. *A clause is a disjunction of literals $\bigvee_j l_j$.*

Definition 3. *A literal is a propositional variable x or its negation $\neg x$.*

Definition 4. *A propositional variable can be assigned a value 1 which represents the boolean value true (positive) or 0 which represents false (negative).*

Definition 5. *An assignment is a mapping from a propositional variable to a truth value. If a mapping exists for all propositional variables the assignment is called complete assignment, otherwise it is called partial assignment.*

Definition 6. *A clause can be in one of the following four states:*

- *satisfied - at least one literal resolves to true*
- *falsified - all literals resolve to false*
- *undefined - at least one literal is unassigned and all assigned literals resolve to false*
- *unit - exactly one literal is unassigned whereas all others resolve to false*

2.2 Incomplete SAT Solving

Incomplete SAT algorithms are designed to find a satisfying assignment in a given time period, but are not able to determine if a problem is unsatisfiable. It just terminates reporting an error that no satisfying assignment can be found. The basis for almost all incomplete SAT algorithms is stochastic local search. Starting back in 1990 two papers by Adorf and Johnston [1] and Minton et al. [34] revealed that applying local search algorithms to large problems (n-queens) performs better than complete (systematic) algorithms. The success of these two applications were the motivation for Selman et al. to develop *GSAT*, the reference algorithm when speaking of incomplete SAT solvers.

The principle of *GSAT* can be explained easily: A random truth assignment for all variables is generated initially. Then the algorithm 'flips' single assignments (inverts the assigned truth value) that promise to satisfy most clauses. This step is repeated until a satisfying assignment is found or a threshold of maximum flips is reached. The whole process, including a new randomly generated initial assignment, is repeated until another threshold is reached.

The decision which assignment to flip is - as already mentioned above - based on a simple local heuristic, namely the number of additionally satisfied clauses. With this heuristic it is possible to move towards a 'better', more likely solution. The drawback of this strategy is that several problems tend to be only satisfiable with a specific assignment whereas the heuristic tends to flip it because it often occurs inverse. Another consideration they made is called *sideways moves*, which describes the case that flipping an assignment only satisfies as much clauses as not flipping the assignment. It was observed that taking *sideways moves* into account yields much better results for many problems. An interesting fact about *GSAT* is that the authors are not able to explain why it performs so well on chosen problems.

There are many optimizations to the basic algorithm of *GSAT*, which actually gain performance by introducing diverse heuristics and additional randomizations (often called *noise* in the literature). A good overview of optimizations on incomplete SAT solving is given in [27].

More information on incomplete SAT solvers can be found in Sect. 5, where *UnitWalk*, an incomplete SAT solver introduced by Hirsch and Kojevnikov in [26] is modified to utilize bit-parallel operations. The modified version *UnitMarch* of Heule and van Maaren has been introduced in [23].

2.3 Resolution-Based Procedure

From this section on this work focuses on complete SAT solvers, where two algorithms build the basis for SAT solvers nowadays – resolution-based and learning-based procedures. The first resolution-based one that actually revealed significantly improved performance over a naive guessing algorithm was the Davis-Putnam procedure (*DP*) [14] and its enhancement by Davis, Loveland and Logemann called DLL-algorithm [13]. *DP* and *DLL* only differ in one point of the algorithm that defines the way the formula is traversed. *DLL* saves main memory by introducing recursion by splitting up the formula in contrast to just adding clauses like *DP* does. The *DLL*-algorithm generally utilizes four rules (the first three are identical to *DP*; names are not exactly original ones):

1. *Unit-clause rule*: A unit clause is a clause that contains only a single literal l. To satisfy the formula this literal has to be assigned to *true*. All other clauses in the formula which contain l are removed, all occurrences of $\neg l$ are also removed.

2. *Pure-literal rule*: If literal l is present in one or more clauses but $\neg l$ is never used, then every clause containing l can be removed.

3. *Eliminating atomic formulas rule*: If a formula is in the form

$$(A \vee l) \wedge (B \vee \neg l) \wedge R \text{ where } \{l, \neg l\} \notin A, B, R$$

where A and B are parts of a clause and R is the rest of a formula (none, one or multiple clauses) it can be replaced by

$$(A \vee B) \wedge R$$
$$\text{or}$$
$$(A \wedge R) \vee (B \wedge R)$$

The first replacement is used by *DP* whereas the second one is used in the *DLL*-algorithm. The advantage of *DLL* is the capability to save memory by splitting the problem and handle the resulting parts separately.

The result of choosing the literal that is used to resolve the formula to new ones is called *decision*. This decision determines how fast a result is found. This was the initial reason why heuristics had to be found to perform well.

4. *Splitting rule*: The resulting clauses from rule 3. are handled recursively one after the other, equivalent to solving two subproblems. One subproblem refers to the assignment l while the other one refers to $\neg l$.

The algorithm applies these four rules as follows: Apply rule 1 until no unit clauses can be found, if no clauses are left, the formula is satisfied. If an inconsistent clause is found the formula is unsatisfied. The next step is to apply rule 2 to eliminate pure literals, if unit clauses are found, step back to applying rule 1 again, then check satisfiability of the formula again.

After all unit clauses and pure literals are resolved, rule 3 is applied (*DLL* version): A literal l of the clause with maximum size is chosen which breaks up the work in two paths. The algorithm is called recursively with first l and, if it returns unsatisfied, $\neg l$. If a recursion path returns the sub-problem to be satisfied this holds for the whole problem and the algorithm terminates.

If both recursive calls on the highest level (the first decision) return unsatisfied, the problem is UNSAT and the algorithm terminates. The process of finding a conflict and backtrack the recursion hierarchy until a level is reached where the inverse is still to be tried is called *chronological backtracking*. From another perspective *DLL* can be seen as a depth-first search, where decisions are the different branches in a binary search tree.

A very important point regarding this work is the following: when applying the first rule that handles unit-clauses, it often happens that another clause in the formula becomes candidate for the rule again. Application of the unit-clause rule until no new unit-clauses are found is called *unit propagation* or *boolean constraint propagation* (*BCP*), a term that was introduced by Zabih in [39]. BCP is described in more detail in Sect. 4.2 as it is the main part of *failed literal probing*.

2.4 Learning-Based Procedure

The next fundamental step to more powerful SAT solvers was done in 1996 by Silva and Sakallah [37]. They developed a new SAT solver called *GRASP* (Generic seaRch Algorithm for the Satisfiability Problem) which analyses conflicts that occur during the search of solutions in a more sophisticated way. The advantage of this conflict analysis is the ability to prune search space more effectively.

When making decisions during the search there are possibly decisions that obviously lead to conflicts, other decisions may never cause the search to fail. If such a conflict occurs the algorithm needs to undo (recover) previously made decision(s) and search in another direction (usually flip a decision to the other truth value). When looking at the decisions that are flipped, it is often observed that they did not really generate the conflict. The originator of the conflict can theoretically be located on every lower decision level.

This is the point where the algorithm introduces *non-chronological backtracking.* Backtracking means to revert decisions that are originators of the produced conflict. To 'remember' what lead to that conflict a so-called *conflict-induced clause* is generated and added to the CNF. This clause is often called *learned clause* in literature.

Knowledge of the originators is obtained by storing the chain of decisions and the corresponding propagations during the search. In case of a conflict, the originators are found by backtracking, starting from the conflicting literal. This technique has emerged the new term of *conflict-driven clause learning* (*CDCL*) in literature.

The implementation of this backtracking feature is done with an implication graph. It is a directed graph that points from every assignment to the clause that contains the reason for assigning it (if the assignment is deduced from unit propagation). A decision assignment therefore has an empty pointer because it does not have a reason. By traversing these pointers it is possible to gain all the information that is required to find the originators on all decision levels and take the proper actions. That is to change the assignment of the causing decision variable or to stop the algorithm because the other polarity also has lead to a conflict (showing that it is UNSAT).

Learning new clauses can be done in various ways, the naive way is to just add all decisions that are found during the implication path traversal. As learned clauses of this form are less-than-ideal because the implications often are 'weak', more sophisticated methods have been developed. Zhang et al. compare different learning schemes in [41]. They revealed that a scheme called *1-UIP* (*first unique implication point*), which is similar to the one employed by *GRASP* [37], yields best results.

2.5 Look-Ahead-Based Procedure

Look-ahead solvers are also based on *DLL* but utilize several additional techniques to be able to solve especially hard problems like random 3-sat formulas. The core of *DLL* is unchanged, only the part where decisions have to be made is improved by a look-ahead algorithm. This look-ahead algorithm tries to find the best decision variable but also simplifies the current formula. Additionally it produces a measure which phase the decision variable should be assigned to find a solution quickly.

These heuristics exploit data gained by the look-ahead part of the algorithm. Every time a new decision variable has to be found it takes the actual assignment state and tests the effects of assigning and propagating other unassigned literals. The heuristics typically are based on the number of additionally satisfied literals, simplified or satisfied clauses. There exist many work about these decision heuristics that can be found in further readings refered to in [24].

The simplification part of the look-ahead algorithm utilizes *failed literal probing* (see Sect. 4). This technique performs *BCP* on both polarities of an unassigned literal and then starts reasoning about the results. One option is that propagation of one polarity leads to a conflict whereas the other polarity doesn't - in this case a new failed literal is found, and the successful polarity can be fixed to that value. This causes a simplification of the formula - clauses will shrink or get satisfied at all.

The look-ahead procedure of this kind of solvers is very interesting to this work since it heavily employs *failed literal probing*. The first full-featured look-ahead SAT solver *POSIT* was developed by Freeman in his PhD thesis in 1995 [17]. Freeman discovered that applying look-ahead for every unassigned literal in every decision step slows down the algorithm. Better performance can be reached by reducing the number of literals in the look-ahead, the deeper the search tree gets (this is because on higher levels the decisions are more significant).

A problem with this technique is that it introduces considerable overhead to the selection of an adequate decision variable which only pays off in case of hard SAT problems. In most cases *CDCL* solvers outperform look-ahead solvers.

More on look-ahead based solvers can be found in a overview paper by Heule and van Maaren [24]. It contains historical informations as well as state-of-the-art techniques and further analysis of heuristics.

3 Parallel SAT Solving

There are different alternatives to make SAT solvers work in parallel, depending on properties like problem partitioning, what kind of architecture the parallel system provides or how information between components of the solver are exchanged. Since the development and use of multi-core systems dramatically increased in the last years, it is obvious that these systems are getting more important. Many current papers focus on exploiting parallelism on multi-core machines where different algorithms can be employed due to the shared memory architecture.

This section will show the main strategies that have been developed to obtain speed-up. What can be said in general is that static partitioning of work is not useful in the area of this kind of search algorithms. This is a straightforward issue because complexity of partitions can be unequally distributed even if work is partitioned equally. Therefore all presented solutions for parallelization are more or less dynamic in their nature.

3.1 Embarrassing Parallelism

This type of parallelism has no complex scientific background, it just uses the fact that in practice multiple SAT problems need to be solved at once. The most straightforward way is to take a bunch of separated SAT problems that have to be solved and process them in parallel. There are different ways to implement a task scheduler to distribute the problems to the workers.

In general there will be a number of workers that utilize a sequential SAT solver to be able to solve arbitrary SAT problems. A master will then manage the distribution of problems that need to be solved, which is just a simple scheduling problem. Since the problems are totally independent from each other there is no need of synchronization between the workers. The only data that needs to be exchanged between master and workers is the problem and the solution.

This approach is often used in practice - not limited to the field of SAT solving. It also scales very well until a point where problems get really hard to solve and therefore take much time while others can be solved faster. Another limiting factor of scalability is the number of provided problems - the more workers solve the same number of problems, the lower the efficiency gets because of inactive workers.

When really large and complex problems have to be solved this solution reaches its limits because there is no way to speed up solving single problems except utilizing more powerful workers.

3.2 Guiding Path

The term *guiding path* was introduced by Zhang et al. in a paper from 1996 where they presented the parallel SAT solver *PSATO* [40]. *PSATO* is an improved parallel version of the sequential SAT solver *SATO* that is based on *DP*. The goal of Zhang et al. was to utilize idle workstations of a computer network with a flexible, robust distributed solver.

A *guiding path* can be described as the state of a running search in the binary tree that is built when decisions are made in *DP*. Edges (links) in the tree consist of two components: the decision literal and a value that shows if it points to the first or second child of the parent node. Zhang et al. distinguish two states an edge can have: *open* if the edge is the first child and *closed* if the edge is the second child. Now it is possible to describe the path from the root to an arbitrary node by a list of edges – this list is called *guiding path*.

The sequential algorithm now is modified to support a guiding path as the second parameter - additionally to the input clauses. This guiding path is used as a current state of the search, and the *open* edges in the path are used when new literals for splitting are needed. Whenever the algorithm is interrupted it is possible to obtain the current state through extracting the *guiding path* of the current search tree again.

Guiding paths became frequently used to split up the work in many SAT solvers and is referred in many publications, including [6, 9, 28, 36]. Splitting work using *guiding paths* nevertheless has the drawback that choosing the split variable randomly may lead to very unbalanced complexity of the subproblems.

To improve the selection of appropriate split variables Martins et al. utilize a technique called *VSIDS* (variable state independent decaying sum) decision heuristics (that has its origins in SAT solver *Chaff* as a technique to find the next decision variable [35]) to determine which variables are relevant in the search tree in a preprocessing phase [31].

There are several other methods to split the search space that are not that popular so they are not present in this work. For further information about alternatives to *guiding path* we refer to a parallel SAT survey paper by Martins et al. [32].

3.2.1 Master-Slave Parallelization

One of the most popular way to utilize *guiding paths* for parallelization is the master-slave model (e.g. [6, 28, 36, 40]). The master handles the dynamic partitioning of the problem. It generates *guiding paths* that are subsequently distributed to requesting slaves that are running a SAT solver instance. These algorithms have several differences in detail but the general behaviour is the same.

Since the work that is necessary to solve a given subproblem is not known in advance, the generation of new guiding paths - or in other words, new subproblems - is handled on demand. Whenever a slave becomes idle because it finished processing a subproblem (proved that it is UNSAT) it requests new work from the master. The master in turn tells another active slave to split his subproblem by generating two separated guiding paths. One path is still handled by the splitting slave, the other guiding path is given to the master which accomplishes the request from the idling slave.

The master now can determine if a problem is SAT or UNSAT. If a slave reports that a subproblem is SAT it can instantly stop all other slaves and terminate with SAT. If all slaves are requesting new work from the master the problem is UNSAT because all subproblems have been proved being UNSAT. This schema of generating guiding paths whenever requested by a slave guarantees that all of them are well utilized through the whole runtime of the algorithm.

3.2.2 Cube and Conquer

Cube and conquer is a technique that combines the advantages of look-ahead-based solvers and *CDCL* solvers. The term was introduced by Heule et al. in a very recent paper in 2012 [22].

Look-ahead solvers employ advanced reasoning which literals are important especially in early stages of the search (more details can be found in Sect. 2.5). *CDCL* solvers are known to perform best on kind of easier but still large problems, because they produce less overhead by employing fast heuristics. However, experiments have shown that combining these two types of solvers can outperform other state-of-the-art parallel solvers on hard instances [22].

The solving process of the cube and conquer solver introduced in [22] is basically split in two phases. The cube phase utilizes a modified version of the look-ahead solver *march* developed by Heule in his masters thesis [25]. This phase results in so-called *cubes*, which are comparable with *guiding paths*.

A cube defines a partial assignment of a formula - additionally the original formula is simplified by common look-ahead techniques like failed literal detection. The most critical quality issue when generating these cubes is to find optimal cut-off-points, that is to stop going deeper in the search tree. The generated cubes should be 'easy enough' to be efficiently solved in the second phase.

Cubes are exported in a special *iCNF* file that can hold multiple formulas, each with the assumptions that have been made in the look-ahead phase. Solving the generated cubes is done in the second phase where a common state-of-the-art *CDCL* SAT solver is used (a modified version of *lingeling* by Armin Biere to support *iCNF files*).

Depending on the original formula it is possible that up to millions of cubes are generated. These cubes can be solved either sequential or parallel, the parallel version is very straightforward since no synchronization is necessary (similar to embarrassing parallelism in Sect. 3.1).

As the experiments in [22] have shown that this approach is able to outperform ordinary guiding path as well as portfolio solvers (see Sect. 3.3) on hard instances. The power of look-ahead solvers to make well-reasoned top-level decisions and the high efficiency of *CDCL* solvers seem to have great potential.

3.3 Portfolio

The idea behind portfolio parallel SAT solvers is to apply diverse sequential algorithms and/or varying parameters to the SAT problem. It has been seen in the SAT competitions[1] that SAT solvers can perform very diverse on different problem domains - depending on their characteristics.

One kind of portfolio solvers utilize this fact to run different sequential solvers in parallel resulting in surprisingly good results. The other type of portfolio solvers run the same sequential SAT solver in parallel but use different parameters for the single instances. This works due to the stochastic nature of SAT problems - different parameters can influence the runtime of the algorithm significantly.

The first parallel portfolio SAT solver was *ManySAT* introduced by Hamadi et al. [20]. It utilizes multiple instances of the same sequential solver (based on *MiniSat* [15]) but every instance has different settings. They use combinations of restart behaviour, decision variable heuristics, polarity of decision variables and clause learning scheme. Depending on how many cores should be used the settings can be varied.

Another technique that is used is clause sharing among the different instances of clauses that have at most eight literals - which is an empirically evaluated value. Experiments on instances of *SAT-Race 2008* revealed that *ManySAT* is able to gain a superlinear speedup of 6.02 when utilizing 4 cores.

The portfolio-based SAT solver called *SATzilla* by Xu et al. [38] follows a slightly different approach. It maintains a portfolio of state-of-the-art SAT solvers that can be employed for solving the problem. SATzilla tries to determine what the runtime of each solver would

[1]`http://www.satcompetition.org/`

be by a component called *approximate runtime predictor*. This information is the basis to decide which SAT solver is used to tackle the problem.

To be able to predict the runtime of solvers on specific problems Xu et al. apply machine learning techniques on previously chosen problem classes and the results of the solvers contained in the portfolio. The results of this machine learning process are so-called *empirical hardness models*. These learned data is then used to identify how well each solver will perform on a given problem.

More details on how analysis of a given problem works (feature extraction to be able to classify a problem) can be found in [38]. What has to be mentioned is that *SATzilla* does not utilize parallelism although there are possible scenarios for parallelism. Nevertheless, since SATzilla is attending the SAT competition (first time in 2003,) it performs very well and often is placed on top. SATzilla even won the recent SAT competition in 2012[2].

The potentially simplest portfolio solver that has been developed is *ppfolio* by Olivier Roussel. There is no scientific paper available which discusses its implementation, but an algorithm description which has been submitted to 2011th SAT competition[3]. *ppfolio* itself does not know anything about solving SAT instances, the only thing it does is to execute other SAT solvers in parallel. Roussel states that *ppfolio* 'is not clever at all' himself. There is a fixed schema of how SAT solver instances are assigned to threads when a given number of threads are available.

Roussel selected the installed SAT solving programs based on the results of SAT competition 2009 (*cryptominisat*, *lingeling/plingeling*, *clasp*, *TNM*, *march_hi*). To everyone's surprise *ppfolio* won 16 medals in SAT competition 2011, which demonstrated that this straightforward approach is very powerful.

3.4 Fine-Grained Parallelism

All parallelization techniques presented so far are high-level approaches. Few publications have been made about low-level parallelization of the core of a SAT solver, namely *BCP* (see Sect. 4.2). Since constraint propagation itself is a sequential problem due to dependencies that arise out of implications it is questionable if it makes sense to parallelize *BCP* at all.

An example of the worst-case CNF formula for parallel *BCP* looks as follows (based on an example in [17]):

Example 1. $(1) \wedge (\neg 1 \vee 2) \wedge (\neg 2 \vee 3) \wedge (\neg 3 \vee 4) \wedge (\neg 4 \vee 5) \wedge ...$

The implication graph for this example has only one starting point (1) and no branches, making it impossible to parallelize anything. The fastest way of propagation is exactly the number of clauses:

Example 2. $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow ...$

This example gives an idea why *BCP* is proven P-complete [17].

[2]`http://baldur.iti.kit.edu/SAT-Challenge-2012/results.html`

[3]`http://www.cril.univ-artois.fr/~roussel/ppfolio/`

Manthey developed an algorithm that enhanced his *CDCL* algorithm *riss* to use multiple threads for *BCP* [30]. The implementation only handles *BCP* with multiple threads, the rest of the algorithm is handled in one single thread to be able to see the effects isolated.

This is how the *BCP* part of the algorithm works: The clauses of the formula are partitioned, every thread obtains an equal portion of clauses which are handled in the following. The main thread decides which clauses are handled by which thread and this threads gains the exclusive write access to the clause. Additionally, the data structures used for unit propagation are maintained by each thread separately (watch lists, propagation queue, trail, reason information). When a literal needs to be propagated, every thread's propagation queue is initialized with this literal and executes propagation on its associated clauses. When propagation on each thread terminates, the found implied literals are exchanged between all threads and propagation in each thread starts over again.

Whenever a thread finds a conflict, all threads stop their work and the main thread starts the *CDCL* conflict analysis. When no new literals can be implied by any thread, unit propagation terminates and the main thread uses the generated information to succeed with *CDCL*.

Experiments revealed that the use of four cores is less effective than just utilizing two cores. With an average speedup of approximately 1.1 the algorithm does not perform very well, but there are optimizations the author thinks of: enhancing locking techniques and dynamic load balancing. It is also imaginable to use this technique to additionally use parallelism in a portfolio solver to be able to utilize more available cores.

Hyvärinen and Wintersteiger [2] take a slightly different approach. Instead of partitioning the clause database as Manthey does they employ continuous synchronization between threads. The threads therefore share a common propagation queue and just request literals to propagate at this global point. This approach implicitly takes load balancing in account, at the cost of more synchronization overhead. To keep this synchronization overhead as small as possible, fast atomic operating system calls are used when possible.

Simulations and experiments revealed that this approach is not able to gain any speedup. The synchronization overhead overcomes the parallel execution of propagation. The main problem is that the propagation queue contains insufficient literals to fully utilize the workers. Several improvements using faster locking techniques are also proposed by the authors.

Another way of exploiting fine-grained parallelism is presented by Zhao et al. in [42]. They developed an application specific multiprocessor system and a new SAT solving algorithm that especially exploits the properties of the underlying hardware. The single processing units support complex operations that are often used in the domain of SAT solvers, a customized message passing system tries to avoid memory bandwidth bottlenecks. The processors are connected to each other in a grid layout to reduce latencies and optimize hardware issues.

Clauses are partitioned (like Manthey did in [30]) and distributed to the processing units, which work asynchronous via message passing and polling techniques. The results of the experiments show that this application specific hardware approach works surprisingly well, at the cost of flexibility and limited problem size. These experiments revealed that a speed-up of up to 60 is possible when utilizing 81 processors - compared to *Chaff*. In general the speed-up is higher for larger problems because of better utilization of the processors.

As the first two examples of low-level parallelization approaches have shown, it is extremely difficult to achieve speed-ups. Hardware solutions are also not the best way to go because further development of commercial processors will outperform them anytime soon. This work tries to apply another kind of low-level technique to parallelize *failed literal probing* which also has *BCP* at its core.

3.5 General Techniques

3.5.1 Clause Sharing

Many parallel solvers share their knowledge of learned clauses via clause sharing techniques. Clause sharing is only relevant for *CDCL* SAT solvers which generate learned conflict clauses through a given learning scheme (see Sect. 2.4). Sharing clauses is interesting especially if *different* parts of the search tree are explored in parallel. Gained information about clauses that cause conflicts can prune large parts of the search tree. Experiments have shown that sharing all learned clauses leads to bad performance due to exponential growth of the clause database, so heuristics and limitations for clauses have been developed [6, 20, 36].

Many implementations just limit the size (the number of literals) of the learned clause to a fixed value. This value is determined through empirical observations. Whenever the size of the clause exceeds the limit it is not shared with the other instances.

Latest versions of *ManySAT* employ a new strategy on deciding which clauses to share [19]. They observed that the size of learned clauses increases while the search process proceeds. This means that with a fixed threshold less clauses are shared via the workers towards the end of processing. The solution to this is a dynamically adjusted clause sharing policy, which is calculated from throughput and quality measurements. This dynamic approach has shown to be more effective than the static one [19].

4 Probing-Based Simplification

SAT problems are often generated or translated from high-level designs utilizing a set of rules, which can be far from optimal. That causes CNF formulas that have a great simplification potential. Even for manually constructed problems great improvements are possible, depending on the structure of the CNF. Observations on solving SAT instances reveal that in many cases simplified formulas - containing less clauses and variables - can be solved faster by SAT solvers.

That's the reason why much effort has been put into developing effective preprocessing techniques for SAT formulas. There are diverse alternatives to discover simplification potential. The most researched techniques are those based on probing.

Probing in this domain is a multi-level procedure. The first step is to make assumptions about the variable assignment. These assumptions are then propagated throughout the formula using BCP. This allows to infer new knowledge out of changes in the assignment (additional variable assignments that are fixed). The generated knowledge is then used to simplify the formula - depending on which levels of simplification to use.

To be able to generate new knowledge a set of rules is applied to the formula that allows interpretation. Possible outcomes of this processing can be detection of congruent variables, detached variables or clauses, or even generation of new clauses that can simplify the search of the following SAT solving process. A problem of these simplifications is that there is no guarantee this processing pays off at all.

SAT solvers utilize these probing-based techniques in different ways. They are often used in a separate program that takes the formula as an input, simplifies it and produces a new output formula. This simplified formula is then passed to an arbitrary SAT solver. There also exist SAT solvers that implement their own preprocessing steps - the preprocessing algorithm can utilize data structures and methods of the core SAT solver. The core SAT solver itself has the opportunity to reuse knowledge generated by the preprocessing step. The last type of SAT solvers to utilize probing are lookahead solvers (see Sect. 2.5). They integrate probing deeply in the search process - in this context this is called probing lookahead.

An overview on probing techniques can be found in papers by Le Berre [4] and Lynce and Marques-Silva [29].

A more practical view on probing-based simplification techniques can be found in a paper by Heule et al. [21]. In addition to introducing a new simplification technique called *hidden literal elimination* (*HLE*), this work integrates various chosen simplification techniques in a state-of-the-art solver which are able to maintain a notable performance gain.

4.1 Basics

Every probing-based simplification on a SAT formula starts with the propagation of every possible assignment of all literals. That is to propagate every possible assignment α through a formula φ until nothing changes (more details on that can be found in Sect. 4.2).

An assignment α is defined by a tuple of a literal l and a value v (which can be 0 or 1 - *false* or *true* respectively), $\alpha = \langle l, v\rangle$.

The result of this propagation is the following: every assignment α has a corresponding set of implied assignments denoted by $BCP(\alpha)$. Please note that the result of $BCP(\alpha)$ also includes the currently propagated assignment α.

Section 4.3 discusses the possible simplifications and a set of rules that are required to derive them.

4.2 Boolean Constraint Propagation

Boolean constraint propagation does not only build the core of probing-based simplification techniques, but also for all algorithms that are based on *DP*. *BCP* is the most expensive task of SAT solving nowadays - with almost 90% of total solver runtime [35]. That is why *BCP* has to be implemented as efficient as possible.

A short recap about the idea of *BCP*: The *unit clause rule* of *DP* describes that a clause that only contains negative assignments and exactly one unassigned literal implies to set this literal to true. Propagation also detects conflicts by implying literal assignments that contradict existing assignments.

There are different ways to determine for which clauses in the formula this rule can be applied. Early implementations were straight-forward and did not use any implementation tricks to find those clauses efficiently. First steps into optimized versions of unit propagation have been made with occurence lists, which utilize additional data structures to reduce the number of clause visits. Current sate-of-the-art solvers use two-watched-literal propagation introduced in [35] as it has shown best results in benchmarks.

4.2.1 Occurrence Lists

The most basic approach to propagate units is to simply iterate over all clauses and calculate the state of each clause. That is to apply the actual assignment to a clause and identify if the clause is a unit clause. If a new unit has been found all clauses are iterated again until no new units are found or a conflict is detected. This is not very efficient because for every unit detected all clauses have to be visited.

An optimization that is widely used is to store *occurrence lists* for each literal (e.g. [7]). Each list stores references to all clauses that contain the regarding literal. These lists can now be used to only iterate over clauses that have the chance to become unit clauses. Clauses that do not contain the newly assigned literal cannot change their state at all. These occurrence lists are also called *watch lists* in literature.

Another alternative to *occurrence lists* is to use a counter that stores the number of undefined literals for each clause. One of the first publications that describe this improvement is published by Crawford et al. in [11] as a method of their SAT solver *Tableau.* The counter is initialized with the number of literals a clause contains when the algorithm starts. Occurrence lists for literals in every polarity are necessary to instantly be able to decide if a clause potentially becomes a unit clause.

When an assignment is propagated, the referring occurrence list is iterated and the counter of each clause is decremented. Whenever a counter becomes 1, a unit clause or conflict is found. The literals of the clause have to be iterated to find the correct literal that needs to get propagated again. This really speeds up propagation because only potential clauses have to be visited and the only operation that has to be done is to decrement and check the counter. The clause only has to be iterated if the counter's value is 1.

A technical report by Biere [5] adds another feature to the counting technique, that replaces the need to iterate over the literals of a clause to find the unit literal. Biere utilizes a field in addition to the counter to store the 'sum' of the clause. This field is initialized with an *XOR* operation over all literals in the clause. Every time the clause is visited the sum field is updated through another *XOR* operation with the actually propagated literal. When the counter's value is 1 the sum field contains the unit. With this trick it is possible to replace many clause iterations with a simple and fast *XOR* operation.

Nevertheless the drawback of using such counters is that if backtracking is necessary and decisions have to be undone, all the counters have to be adjusted.

4.2.2 Two-Watched-Literal Propagation

Two-watched-literal propagation was introduced by Moskewicz et al. in [35] when they developed a new SAT solver called *Chaff*. They asked themselves if it is necessary to know the exact number of undefined literals in a clause because the only interesting state is when switching from two to one undefined literals - that is the case when the clause needs detailed inspection.

To accomplish this idea they initially pick arbitrary two (undefined) literals out of every clause and make them the *watched literals* for the corresponding clause. When a literal is propagated, only two literals of each clause can match the inverted propagation literal which is potentially much less for longer clauses than with full occurrence lists. If a watched literal is hit for a clause there are two possible outcomes:

1. The clause contains another undefined literal, additionally to the two already watched literals. To fulfil the constraint that the two watched literals must be undefined, the hit watched literal is exchanged with the additionally found undefined literal. Now that the two watched literals are undefined again it is guaranteed that the clause cannot be unit until another watched literal is hit.

2. The clause does not contain undefined literals except the ones that were watched already. That means that a unit clause has been found, and the unit literal is the second, non-hit, watched literal which can easily be retrieved.

Another benefit of this watching scheme is that nothing needs to be done when backtracking, in contrast to the counter-scheme where the counters need to be updated to remain consistent.

4.2.3 Bit-Parallel Propagation

This thesis evaluates the potential of an algorithm that can handle multiple literals in every propagation step using bit-parallelism. The development of modern CPUs not only heads to multiple cores but also to support more complex instructions including more bits. In more detail these are SIMD (Single Instruction Multiple Data, [16]) styled operations that can apply the same operation to multiple pieces of data in parallel. Latest Intel® processors support the new instruction set *AVX2*[4] which can handle shift and the most common logical operations on bit-vectors with sizes up to 256 bits. As described later in the implementation details it is possible to handle up to 128 literals in parallel because a single literal assignment is encodable in two bits.

There exists earlier work by Heule and van Maaren that utilizes bit-level operations [23]. They modified the existing local search SAT solver UnitWalk [26] so that unit propagation is handled bit-parallel. More on related work is found in Sect. 5.

Details on the algorithm that has been implemented and the used data structures and operations can be found in the implementation Sect. 6.

4.3 Reasoning

This section contains rules that define how results of *BCP* have to be interpreted to obtain new knowledge.

There are two options on how the formula can be altered. The first one is to deduce assignments that can be fixed to a value or the value of another literal. The second is to deduce new clauses which may simplify solving a formula by introducing new implications.

The following subsections will review the findings by Le Berre [4] and Lynce and Marques-Silva [29]. For every single finding the corresponding theorem or proposition of the two papers are given.

Le Berre also deals with *double* unit propagation lookahead where two assignments are propagated. That way it is possible to do more complex reasoning and find more biconditionals. This kind of probing is not in the scope of this thesis, so for further information we refer to the paper by Le Berre [4].

4.3.1 Deducing Assignments

This first deduction section defines rules to obtain literal assignments that can be fixed due to reasoning. Two of them are often used in look-ahead based solvers - namely called *failed literal* and *lifting rule.*

1. *failed literal rule* The failed literal rule says that if an assignment leads to a conflict when applying *BCP*, the opposite value needs to be assigned to be able to satisfy the formula.

   Definition 7. $\perp \in BCP(\langle x, v\rangle) \Rightarrow \varphi \models \langle x, \neg v\rangle$

   This rule refers to theorem 2.3 in [29] and proposition 2 in [4]. The first one to call this technique *failed literal probing* was Freeman in [17]. This is actually the most often used probing technique at the moment.

[4]Advanced Instruction Set 2 – `http://software.intel.com/file/36945`

2. *lifting* The lifting rule says that if *BCP* of both polarities of a literal results in same assignments these assignments can be fixed.

   Definition 8. $\langle y, 1\rangle \in BCP(\langle x, 0\rangle) \cap BCP(\langle x, 1\rangle) \Rightarrow \varphi \models \langle y, 1\rangle$

   Le Berre defines this rule in proposition 3 and Lynce in theorem 2.1.

3. All literals x_j of a clause c_i are assigned a value to be satisfied. If applying *BCP* to these assignments and the intersection of the results contains new assignments, they can be fixed.

   Definition 9. $\langle y, 1\rangle \in \bigcap_{x \in c_i} BCP(\langle x, 1\rangle) \Rightarrow \varphi \models \langle y, 1\rangle$

   This rule is only present in Lynce's work [29] as theorem 2.2.

4.3.2 Deducing Clauses

This section contains rules to generate new clauses out of reasoning. These new clauses may simplify later SAT solving significantly by exposing important implications.

1. *implied (obvious) binary clauses* This first clause rule describes a way to add the simplest clauses that can be found by *BCP*. Every new literal assignment that is found during *BCP* gives an implication of the propagating assignment to the implicating one.

   Definition 10. $\langle y, 1\rangle \in BCP(\langle x, 1\rangle) \Rightarrow \varphi \models x \rightarrow y$ and $x \rightarrow y \Leftrightarrow \neg x \vee y$

   This definition is given in theorem 2.4 and proposition 1. Le Berre additionally introduced proposition 4 which defines the rule to detect equivalent literals. This proposition just uses the fact that Def. 10 can be applied in both directions - and therefore it is obvious, too:

   Definition 11. $\langle y, 1\rangle \in BCP(\langle x, 1\rangle) \wedge \langle x, 1\rangle \in BCP(\langle y, 1\rangle) \Rightarrow \varphi \models x \leftrightarrow y$

   Another way to deduce equivalent literals is to utilize the following rule. The idea is that if $x \rightarrow y$ and $\neg x \rightarrow \neg y$ they have to be equivalent.

   Definition 12. $\langle y, 0\rangle \in BCP(\langle x, 0\rangle) \wedge \langle y, 1\rangle \in BCP(\langle x, 1\rangle) \Rightarrow \varphi \models x \leftrightarrow y$

2. When propagating both polarities of a literal, every combination of the implied sets of assignments that are generated introduce a new clause.

   Definition 13. $\langle y, 1\rangle \in BCP(\langle x, 0\rangle \wedge \langle z, 1\rangle \in BCP(\langle x, 1\rangle) \Rightarrow \varphi \models y \vee z$

   This rule is defined by Lynce as theorem 2.5. To explain this one just imagine that $\neg x$ implies literal y and x implies z. As either x or $\neg x$ has to be satisfied it is clear that either y or z also have to be satisfied - resulting in the new clause. There can be more implied literals for sure, every combination of the two sets potentially defines a new clause.

3. *hyper-binary resolution* Hyper-binary resolution basically defines a way to overcome multiple implication steps that would be necessary to imply a useful clause.

   For every clause c_i of formula φ there exist following clauses: the cross-product of all $BCP(x_{i_j})$ whereas $x_{i_j} \in c_i$ and $j \in 1, ..., |c_i|$.

   Definition 14. $\forall_{c_i \in \varphi}, x_{i_j} \in c_i \Rightarrow \varphi \models \prod_{j=1}^{|c_i|} BCP(\langle x_{i_j}, 1\rangle))$

   This definition refers to theorem 2.6 in Lynce's work.

   More information on hyper-binary resolution can be found in a paper by Bacchus [3]. This work also includes a slightly different definition of how new clauses can be identified.

4. *assignments causing conflicts* If two different literal assignments imply the same literal in opposite polarities it can be implied that they must not be satisfied simultaneously.

   Definition 15. $\forall x, y \in L(\varphi)((\langle z, 0\rangle \in BCP(\langle x, 0\rangle) \wedge \langle z, 1\rangle \in BCP(\langle y, 0\rangle)) \Rightarrow$
   $\varphi \models x \vee y$
   which uses this implicit intermediate step: $x \vee y \Leftrightarrow \neg(\neg x \wedge \neg y)$

   This rule refers to theorem 2.7 from Lynce - as there can be observed this rule is kind of the opposite of clause deduction rule 2. The difference is that this one utilizes the fact that a clause would get falsified - and rule 2 corresponds to satisfying a clause.

5. *assignments causing clauses to get falsified* Whenever a set of assignments and its resulting implications lead a clause of formula φ to be falsified, one of the initial assignments have to be satisfied.

   Definition 16. $x_1, x_2, ...x_n \in L(\varphi), A = \bigcup_{i=1}^{k}(BCP(\langle x_i, 0\rangle)$
   if $c \in \varphi$ and $c(A) = \perp \Rightarrow \varphi \models x_1 \vee x_2 \vee ... \vee x_n$

   Lynce describes this rule in his work [29] in theorem 2.8.

4.4 Practical Application

As shown in Sect. 4.3 there exist many rules to simplify existing formulæ. In practice this all looks slightly different - many implementations pick out few because not all of them pay off the additional work.

The most often used techniques in SAT solvers nowadays include the *failed literal rule* (Def. 7 and also in [17, 25], as well as a main component of look-ahead solvers - see Sect. 2.5), *lifting* (Def. 8), *equivalent literal detection* (Def. 11) and *hyper-binary resolution* (Def. 14).

This work focuses on the *failed literal rule* for reasons that have already been told in the motivation (Sect. 1.1). There is a failed literal reference algorithm provided by the advisor of this work, Prof. Armin Biere. It is used for verification of the developed algorithm but also used in the evaluation for comparison issues.

Theoretically all of these reasoning rules could be utilized in the algorithm - but this is part of future work that has to be done.

5 Related Work On Bit-Parallelism

5.1 Local Search Parallelization

Heule and van Maaren developed an algorithm based on the existing incomplete SAT solver *UnitWalk* by Hirsch and Kojevnikov [26]. Their work [23] is based on following idea: why not utilize a processors ability to simulate 1-bit operations (boolean operations) on 32-/64-bit processors. Since incomplete SAT solvers do not utilize complex reasoning algorithms they are particularly suitable for this kind of parallelization.

As already said, *UnitWalk* is an incomplete SAT solver (see Sect. 2.2). What is special about *UnitWalk* is that it does not use counting heuristics like most of the other algorithms do - instead it uses *boolean constraint propagation* (Sect. 4.2) to flip variable assignments.

The modified algorithm in [23] is called *UnitMarch*. The *BCP* part of the algorithm generally works like this:

- Every assignment is represented as a 2-bit value, due to the possible states that have to be depicted: *true*, *false* and *undefined* (another optional state is unused at the moment, namely *conflicting*).
- These 2-bit values are split up in two different arrays, one storing the first bit - representing a positive assignment, and the other one stores the second bit - representing a negative assignment.
- These two arrays combined with the clause definitions can be used to find unit clauses. This happens by calculating bitmasks of the state of a clause - similar to the approach used in this work.

They also introduced an additional technique to avoid the phenomenon of multiple identical states in different positions of the vector, which has been observed to occur at a high probability [23]. This technique basically utilizes assignment matrices to detect duplicates. Calculation of these matrices can be very time-consuming in theory but it came out that this fact can be neglected in practice.

Experiments in [23] have shown that even the 1-bit version of *UnitMarch* is comparable in performance with *UnitWalk*. If the algorithm utilizes up to 32 bits in parallel, solution time is reduced dramatically - which is also the case for the number of periods used to find a solution.

6 Implementation

6.1 Input Data

Almost all SAT solvers support the DIMACS CNF format, which is more or less a de facto standard for the storage of boolean formulas in conjunctive normal form (CNF). A DIMACS file basically consists of a header line that begins with the fixed string 'p cnf' followed by the number of variables and clauses that are used within the formula.

Each of the succeeding lines represent a single clause, which consists of one or more, possibly negated, literals and a terminating zero. Positive literals are denoted as positive numbers whereas negated literals have the minus-sign '-' in front of the literal number.

A simple example of a boolean formula in DIMACS format could look like this:

```
p cnf 3 2
1 -3 0
-1 2 3 0
```

This file is a representation for the mathematical boolean formula shown in Ex. 3.

Example 3. $(1 \lor \neg 3) \land (\neg 1 \lor 2 \lor 3)$

6.2 Data Structures and Encoding

6.2.1 Formula Storage

The formula consisting of a specified number of clauses is stored in one block of dynamically allocated memory. It is implemented as a stack that is enlarged if the formula gets too large. Each clause is terminated with a zero-literal in memory to denote the end of a clause, an additional array stores the starting point of each clause via pointer references.

A single literal isn't encoded as a standard signed integer number in memory but uses the two least significant bits to encode the three possible states of a literal assignment:

Table 1: Bit encoding of possible literal states

encoding	literal state
00	positive
11	negative
01/10	undefined

The types used for literals in the implementation therefore are always unsigned ones. The programmatic advantages of this encoding will be handled in a later part of this thesis.

6.2.2 Assignments Storage

To store the assignments states when propagating a certain literal a fitted data structure is required that can represent this. In this algorithm it is basically an array of a specific datatype. The length of this array depends on the number of literals used in the formula, so each literal has one corresponding entry in the assignments array.

As already mentioned before, one assignment has only three different states: *positive*, *negative* and *undefined*. It is straightforward to see that these three states can easily be encoded in two bits that could represent 4 states.

In the base version of the developed algorithm, one element of the assignments array only contains exactly one assignment for the actual propagated literal. If using an unsigned integer as the preferred data type which can store 32 bits and only two bits are used 30 bits get wasted. Theoretically it is possible to store 16 assignments per integer or, if using SSE[5] data types up to 128 assignments (256 bits). To make use of multiple assignments per element the algorithm has to be modified to operate on bit-vectors instead of simple branching behaviour. More information about the use of these vectors is discussed further down.

6.2.3 Advantages of Sign Encoding

An operation that is used very often is to find the actual value of a certain literal. If the given literal is a negative one, the actual assignment has to be negated to represent the actual value. If no special encoding would be used, an *if*-clause gets necessary to decide whether the literal is negative and flip the assignment. With the introduced encoding it is possible to calculate the assignment using a simple *XOR*-operation. The correctness of this statement can simply be shown in a truth table:

Table 2: Assignment sign calculation

literal-sign	assignment	result-sign (*XOR*)
00 - positive	00	00
00 - positive	11	11
00 - positive	01, 10	01, 10
11 - negative	00	11
11 - negative	11	00
11 - negative	01, 10	10, 01

Even if this was a step to make the improved algorithm branch-less, it is also used in the base algorithm to keep differences between the two algorithms to a minimum.

6.3 Base Algorithm

The first step of this thesis was to implement the very basic algorithm of failed literal probing without any optimizations. Other implementations e.g. make use of literal occurrence lists, but those would only complicate the basic idea of making the base task of literal propagation bit-parallel.

In the following, the pseudo-code found in Alg. 1 is explained in more detail.

The only parameter given is the formula which is the parsed representation of the problem in CNF. As a first step, the global assignment array is initialized to show every literal to

[5]Streaming SIMD Extensions; additional instructions for CPUs to apply one operation to multiple junks of data.

Algorithm 1 Basic Failed Literal Probing

```
    function BasicFLP(formula)
2:      initialize global assignment
        repeat                                ▷ repeat until global assignment unchanged
4:          repeat              ▷ repeat until all literals propagated and no new units found
                if first round or assignment unchanged then
6:                  initialize next assignment              ▷ clone global assignments plus
                                                             ▷ the next propagation literal
8:              for all clause in formula do                          ▷ iterate over formula
                    reset flags
10:                 for all literal in clause do
                        if clause not satisfied then
12:                         actass ← assignment[literal]  ▷ get assignment of actual literal
                            if actass is positive then                ▷ clause is satisfied
14:                             set satisfied flag
                            else if actass is undefined then
16:                             if unit_candidate flag set then           ▷ already found a
                                    set invalidate_unit_candidate flag     ▷ unit candidate
18:                             else                                ▷ unit candidate found
                                    set unit_candidate flag
20:                                 found_unit ← literal
                    if satisfied or invalidate_unit_candidate flag set then
22:                     no new knowledge
                    else
24:                     if unit_candidate flag set then                  ▷ new unit found
                            assignment[literal] ← found_unit
26:                     else
                            globalass ← global assignment of propagation literal
28:                         if globalass is undefined then           ▷ failed literal found
                                add inverse propagation literal to global assignment
30:                         else if globalass = propagation literal then ▷ conflict detected
                                return formula inconsistent
32:         until all literals probed and no further unit found
        until no new failed literals produced
34:     return global assignment                      ▷ return the found failed literals
    end function
```

be unassigned. When the algorithm finishes, it contains the computed failed literals of the CNF. The following outer loop is repeated until no more new failed literals are found in the inner loop. The inner loop is executed until all literals have been propagated and no further units can be retrieved. The order of literals to propagate is chosen to be ascending from *1* to m, where m is the number of literals, alternating positive and negative assignments. A sample series for a CNF with four literals shows as follows:

Example 4. $[1, -1, 2, -2, 3, -3, 4, -4]$

Every time no further unit can be found for an assignment the next literal is propagated. This is done by cloning the global assignment to the actual one. The next literal to propagate is also set to the actual assignment, which completes the initialization of the next propagation round.

The detection of new units and failed literals is basically done by setting flags that indicate the state of a clause. The possible states are as follows:

- *satisfied* $\rightarrow$ if the clause contains at least one positive assignment for a literal, it is satisfied.
- *unit candidate* $\rightarrow$ if the clause is not satisfied yet, and an undefined assignment is found, it may be a unit, so the literal is a unit candidate.
- *invalidate unit candidate* $\rightarrow$ if an undefined assignment is already found (a unit candidate is found), and another undefined assignment is found, the unit candidate has to be invalidated.

From these three flags one can deduce if a new unit clause is found or if a clause is not satisfiable any more and the propagated literal is a failed literal. A new unit is therefore derived when the clause is not satisfied, a unit candidate has been found and the unit candidate has not been invalidated. A failed literal has been found if none of the three flags is set, which means that all assignments in the clause lead to negative values.

6.4 Bit-parallel Branch-less Algorithm

The fundamental form of the basic algorithm also applies to the bit-parallel algorithm. It also iterates over the formula with the containing clauses and literals over and over again and calculates clause states. The large difference is how information about the clause state is computed because ordinary *if-then-else* cascades are not applicable to bit-vectors any more. For this purpose complex bit-operations are introduced that calculate the state of a clause only by combining the base operations *NOT*, *AND*, *OR*, *XOR* and *SHIFT*.

The second challenge is to get the main part of the algorithm branch-less to avoid mispredicted branches. Those mispredicted branches result from the CPUs instruction pipeline that has to introduce stalls if the execution pipeline contains incorrect assignments. Due to the immense number of iterations over the clauses there is a large potential to save execution time.

6.4.1 Bit-Vectors

A bit-vector in the case of this thesis always means the array of bits of a simple data-type such as *integer* or *__m128i* (SSE 128 bit integer data-type). The advantage of using simple data-types is the ability to directly apply the basic bit-operations. This also applies to SSE's more complex data types.

Every bit-vector in the algorithm is split into 2-bit entities, which means that a 32 bit datatype contains 16 units. For the implementation one has to choose with which data-type to work with. In the resulting implementation the bit-vectors range from 4 to 128 bits.

The bit-vector data-type is used to store different elements which are related to each other and are used to calculate states. These elements are:

- global assignments
- local assignments
- flags (satisfied, unit candidate, invalidate unit candidate, changed, failed, unit)
- masks (first bit of every unit set, all zeros, all ones)

Assignment bit-vectors have the already mentioned 2-bit encoding (see Table 1) for *undefined*, *positive* and *negative*.

6.4.2 Operations

To be able to handle multiple states of assignments without many loops to iterate over bit-masks it is necessary to define several more complex operations. These consist of well-known logical operations that are available in almost every programming language and can be mapped easily to fast hardware-specific operations. In this thesis symbols based on the *C* programming language are used. The required basic bit-operations include:

- *NOT* (symbol ~): inverts the value of a bit, zero becomes one and vice versa
- *AND* (symbol &): only leads to one if both corresponding bits are also one
- *OR* (symbol |): leads to one when not all two bits are zero
- *XOR* (symbol ˆ): leads to one if either the first or second bit have value one
- *SHIFT-RIGHT* (symbol ≫): shift every bit x positions further to the right inside the bit-vector, the least significant bit is dropped out the vector and the most significant bit is filled up with a zero

Assignment Vector One needs to know what the state of an assignment is, more concrete the least significant bit (LSB) should contain the truth value of the corresponding operation. Equations 1, 2 and 3 show the formulæ to get the necessary information about an assignment - if it is undefined, positive or negative for an arbitrary number of entities.

Equation 1. $undefined = (vector \;\hat{}\; (vector \gg 1))\ \&\ mask\_firstbit$

Equation 2. $positive = \sim(vector \mid (vector \gg 1))\ \&\ mask\_firstbit$

Equation 3. $negative = (vector\ \&\ (vector \gg 1))\ \&\ mask\_firstbit$

At the time of thesis finalization a trivial optimization idea appeared, which is inspired by related work of Heule and van Maaren [23] who applied the identical technique. One could easily store the first bit of an assignment entity in one bit-vector while the second bit is stored in a separate, second bit-vector. This way all shift-operations that have to be done at the moment can be omitted.

The second advantage of saving the bits separately is that twice the number of assignments can be handled in an iteration. This change has no effect on the other state bit-vectors because they only need one bit to represent their state - the actual implementation wastes the second bit that is available.

Unfortunately this optimization did not made it into this thesis and must be postponed to future work.

Clause States As in the basic algorithm, clause states have to be calculated, but this time without branches. Equations 4, 5 and 6 show how the three flags can be computed for multiple entities with bit-operations only.

Equation 4. $satisfied \mid= positive(actual\_assignment)$

Equation 5. $unit\_candidate \mid= undefined(actual\_assignment)$

Equation 6. $invalidate\_unit\_candidate \mid= unit\_candidate\ \&\ undefined(actual\_assignment)$

Another challenge arises when the found units should be assigned to the local assignment. In the branching version the three computed flags only had to be checked with if-clauses and decided what to do. To make this code branch-less intermediate results have to be computed at the end of each clause. Those helper bit-vectors are calculated as defined in Eq. 7, 8 and 9.

Equation 7. $unit = \sim satisfied\ \&\ \sim invalidate\_unit\_candidate\ \&\ unit\_candidate$

Equation 8. $changed \mid= unit\ \&\ \sim failed$

Equation 9. $failed \mid= (satisfied \mid unit\_candidate)\ \hat{}\ mask\_firstbit$

With these vectors and another iteration over the clause one can calculate a new assignment for each literal. As shown in Eq. 10 in the first two lines a mask is calculated that is true for all assignments that are undefined and refer to a new unit (the LSB from the entity is cloned to the second bit by a left shift and an or to get a full mask). The third line removes all bits from the actual assignment where a new literal should be assigned. In the fourth step the literal sign is converted to a bit-vector (the sign at the two LSB is cloned to every entities position) and masked with the *undefinedunit* mask. The result is then logically linked by an or with *newass* which results in the new local assignment.

Equation 10.
$$\begin{aligned}
&undefinedunit = undefined(actass)\ \&\ unit\\
&undefinedunit \mid= undefinedunit << 1\\
&newass = actass\ \&\ \sim undefinedunit\\
&newass \mid= undefinedunit\ \&\ sign\_to\_vector(literal\_sign)
\end{aligned}$$

The last step that is missing is detection of failed literals. For that purpose the failed bit-vector has been calculated before, but this one is only handled if all assignments have not changed in the last iteration. The reason for handling this in normal branch style is that no method has been found to implement this in a branch-less way. So the solution is to just iterate over the bit-vector and handle every entity separately, check if it is failed, if it must be set inconsistent or assign a new failed literal.

6.4.3 Pseudocode

The pseudocode of the branch-less bit-parallel algorithm is shown in Alg. 2. Whenever a more complex operation is used, the comment contains a reference to the equation where the detailed calculation can be found.

The initialization statement clones the global assignment to the local one and sets the propagation literals for the next iteration. The way propagation literals are set to the bit-vectors is illustrated in a further section where the assignment storage strategy is explained (see Sec. 6.5.3).

6.4.4 Example

In the following sections the bit-parallel branch-less algorithm is applied on a simple example problem. The example utilizes 4 bits per assignment, that means two assignments are handled in parallel. The CNF of the problem looks like this:

Example 5.

```
p cnf 2 2
1 2 0
1 -2 0
```

Initialization The first step is to initialize the global assignment with all literals undefined, that is:

Table 3: Global assignments initialization

literal	assignment
1	[01, 01]
2	[01, 01]

The initialization of the local assignments is done in two steps: first, all assignments from the global assignment are cloned. The second step is to assign the literals that should be propagated to the corresponding entries in the assignments bit-vector. In this example four bits per vector are used which results in two assignments in parallel. In the first propagation round literals *1* and *-1* are assigned:

Algorithm 2 Branchless bitparallel FLP

```
function BlBpFLP(formula)
    initialize global assignment
    repeat                                  ▷ repeat until global assignment unchanged
        reset propagation literal list
        repeat               ▷ repeat until all literals propagated and no new units found
            if first round or assignment unchanged then
                initialize next assignment                  ▷ clone global assignment plus
                                      ▷ the next propagation literals (detail in Alg. 3)
                reset failed bit-vector
            reset changed bit-vector
            for all clause in formula do                             ▷ iterate over formula
                reset satisfied, invalidate_unit_candidate, unit_candidate bit-vectors
                for all literal in clause do
                    actass ← assignment[literal]
                    calculate satisfied, invalidate_unit_candidate,
                        unit_candidate bit-vectors                   ▷ according to Eq. 4-6
                calculate unit, changed, failed bit-vectors          ▷ according to Eq. 7-9
                for all literal in clause do
                    actass ← assignment[literal]
                    calculate newass bit-vector                      ▷ according to Eq. 10
                    assignment[literal] ← newass  ▷ assign new unit to local assignment
            if not changed and failed then           ▷ check if all assignments unchanged
                                                                ▷ and at least one failed
                for all failedentity in failed do
                    propagation_literal ← get propagation literal of failedentity
                    if propagation_literal valid and failedentity is set then
                        globalass ← global assignment of propagation_literal
                        if globalass = propagation_literal then
                            return formula inconsistent
                        else if globalass is undefined then
                            add inverse propagation literal to global assignment
        until all literals probed and no new unit found
    until no new failed literals produced
    return global assignment                         ▷ return the found failed literals
end function
```

Algorithm 3 Initialization of assignment vector

```
    procedure InitializeAssignment
2:      if propagation literal left then
            for all literals do                                   ▷ clone global assignment
4:              $local\_assignment[literal] = global\_assignment[literal]$
            for $i$ in $vector\_entity\_positions$ do             ▷ set new propagation literals
6:                                                                ▷ to associated vector position
                $assignment\_vector = local\_assignment[propagation\_literal]$
8:              $assignment\_vector[i] = sign(propagation\_literal)$
                $assignment\_propagation\_literal[i] = propagation\_literal$
10:             if propagation literal left then
                    increment propagation literal
12:             else
                    break
14: end procedure
```

Table 4: Local assignments initialization (1, -1)

literal	assignment	
1	[11, 00]	*11 means negative assignment, 00 means positive one*
2	[01, 01]	*all other assignments are not modified from global one*

Due to be able to calculate the states correctly the bit-vector states are also initialized:

Table 5: State-vectors initialization

flag	assignment
satisfied	[00, 00]
invalidate_unit_candidate	[00, 00]
unit_candidate	[00, 00]
unit	[00, 00]
changed	[00, 00]
failed	[00, 00]

Propagation Now that the assignments and states are initialized, the propagation can start. This happens by iterating over the clauses and literals of the formula, the next table shows how the flags change when iterating over the first clause:

Table 6: First iteration of clause 1 to identify unit clauses

flag	initial	after literal 1	after literal 2	after clause
satisfied	[00, 00]	[00, 01]	[00, 01]	
invalidate_unit_candidate	[00, 00]	[00, 00]	[00, 00]	
unit_candidate	[00, 00]	[00, 00]	[01, 00]	
unit	[00, 00]			[01, 00]
changed	[00, 00]			[01, 00]
failed	[00, 00]			[00, 00]

The new knowledge on the clause can now be applied to the local assignment. One new unit literal in the first local assignment has been found, namely -1. Now the second clause iteration starts to calculate the new assignment using Eq. 10:

Table 7: Second iteration of clause 1 to update local assignments

flag	initial	on literal 1	on literal 2
unit	[01, 00]		
undefinedunit		[00, 00]	[11, 00]
local assignment literal 1	[01, 01]	[01, 01]	
local assignment literal 2	[01, 01]		[**00**, 01]

What can be observed now is that in the local assignment of literal 2 one undefined assignment changed to a positive assignment (highlighted bold).

The same calculations are done with the second clause which results in no new units. After another round that does not result in new units, the propagation ends and the failed literals are handled. In this case no failed literals are found at all (state bit-vector failed is all zero), so the next literals are propagated: 2, -2. Initialization of local assignment looks as follows:

Table 8: Local assignments initialization (2, -2)

literal	assignment
1	[01, 01]
2	[11, 00]

The resulting flags for the propagation in clause one look as follows:

Table 9: First iteration of clause 1 to identify unit clauses

flag	initial	after literal 1	after literal 2	after clause
satisfied	[00, 00]	[00, 00]	[00, 01]	
invalidate_unit_candidate	[00, 00]	[00, 00]	[00, 00]	
unit_candidate	[00, 00]	[01, 01]	[01, 01]	
unit	[00, 00]			[01, 00]
changed	[00, 00]			[01, 00]
failed	[00, 00]			[00, 00]

After calculating and assigning the new unit 1 to the local assignment with propagation literal -2, the new local assignment looks like this:

Table 10: Local assignment after second iteration of clause 1

literal	assignment
1	[00, 01]
2	[11, 00]

Now the second clause is processed:

Table 11: First iteration of clause 2 to identify unit clauses

flag	initial	after literal -1	after literal 2	after clause
satisfied	[00, 00]	[00, 00]	[00, 01]	
invalidate_unit_candidate	[00, 00]	[00, 00]	[00, 00]	
unit_candidate	[00, 00]	[00, 01]	[00, 01]	
unit	[00, 00]			[00, 00]
changed	[00, 00]			[00, 00]
failed	[00, 00]			[01, 00]

After the second clause a failed literal is found with propagation literal -2. A new unit has been detected in the first clause as well, so the formula has to be iterated again to guarantee there are no units missed in propagation. After that, the failed literals are handled: -2 failed to propagate, so literal 2 is assigned positive in the global assignment (for every 2-bit entity in the whole vector):

Table 12: Global assignment after detection of failed literal -2 in clause 2

literal	assignment
1	[01, 01]
2	[00, 00]

The outer loop of the algorithm says that all literals have to be propagated again if a new failed literal is found. This is because clauses processed before now may raise further units or failed literals. In the case of this example, no more units or failed literals are found and therefore the algorithm terminates with the result that literal 2 can be permanently assigned to true, and all clauses containing literal 2 can be removed from the formula. Additionally every occurrence of literal -2 can be removed from all clauses. The resulting simplified CNF for this example is as follows:

Example 6.

```
p cnf 1 1
1 0
```

6.5 Additional improvements

There are several ways to improve the performance of the branch-less algorithm. As already mentioned before the easiest way to gain efficiency is to increase the number of bits used in the assignment vector. With this optimization it should be possible to gain almost linear speed-up because no additional overhead is added. This statement of course only holds until more complex data-types and operations like SSE are used, which introduce overhead for execution or the memory bandwidth is exhausted.

Another popular way to gain performance is to utilize multi-threading, which is also possible for the presented algorithm. There are different methods to parallelize algorithms, the developed algorithm implements a worker approach.

A performance factor for almost every memory-intense algorithm is how well caches can be utilized. An approach to accommodate this is to handle more assignments sequentially while iterating over the clauses to exploit the proximity of the assignments. This should scale well until a hardware-specific limit is reached which is an empirical value depending on the cache sizes and the length of the bit-vector.

Another significant issue with caches is to realize efficient assignment storage, so that accessed data is as local as possible. During the development process considerable optimization potential has been found to change assignment storage to establish a performance gain.

The following sections describe the optimizations that have been implemented for the branch-less version of the algorithm. In the experiment Sect. 7 these optimizations and their effects are evaluated.

6.5.1 Long Bit-Vectors using SSE Instructions

Since the development of the algorithm was done on a machine with 64 bit architecture bit-vectors with 64 bits length can be used natively. Almost every modern desktop and server CPU since 2004 supports the SSE2[6] instruction set which includes logical operations on integer values with up to 128 bits.

[6]Streaming SIMD Extensions 2; http://software.intel.com/sites/default/files/m/9/4/c/8/e/18072-347603.pdf (retrieved on 2012-09-12)

The implementation of the algorithm allows to easily switch between different vector sizes. This is done with a preprocessor setting called *VECTOR_SIZE* which defines which operations and types should be used. As shown in List. 1 the basic code can stay the same applying user-defined types and macros for the bit-operations.

```
#if VECTOR_SIZE == 128
typedef unsigned long long VectorBaseType;
typedef __m128i AssignmentVector;

#define _OR(a, b) _mm_or_si128(a, b)
#define _AND(a, b) _mm_and_si128(a, b)
#define _XOR(a, b) _mm_xor_si128(a, b)
#define _SHR(a, c) _mm_srli_epi64(a, c)
#define _SHL(a, c) _mm_slli_epi64(a, c)
#define _NOT(a) _mm_andnot_si128(a, MASK_ALLONES)

#elif VECTOR_SIZE == 64
typedef unsigned long long VectorBaseType;
typedef unsigned long long AssignmentVector;

#define _OR(a, b) (a | b)
#define _AND(a, b) (a & b)
#define _XOR(a, b) (a ^ b)
#define _SHR(a, c) (a >> c)
#define _SHL(a, c) (a << c)
#define _NOT(a) (~a)
```

Listing 1: Vector-size dependent Types and Macros

The *__m128i* data-type and *_mm_** operations are the SSE2 intrinsics that can be used by including the corresponding header file *emmintrin.h.* The source code contains several passages where special treatment for SSE2 data types is required, e.g. debug outputs.

6.5.2 Parallelization/Multithreading

The use of multi-threading is implemented very straightforward: a worker pool is created on startup, the number of workers is given as a parameter. The initial thread is the 'boss'-thread which coordinates starting and joining the other worker-threads. The worker-threads are started round-wise, that is to propagate every literal once.

The workers are started in rounds. One round means that all literals are propagated once, resulting in a list of failed literals. If no new failed literals are found in a round the algorithm terminates, otherwise a new round is started. One central method controls which worker propagates which literals, this is achieved through a globally locked variable that stores which literals to propagate next.

The technology used for parallelization is *Pthreads*[7] which offers a simple way of spawning workers and managing access to global resources. Correct variable access management is obtained by provided locking methods.

As shown in List. 2 a worker is a very simple structure. The most important component for threading itself is the *pthread_t* variable. The *id* is used for debugging purposes to identify which thread is performing which actions and to identify the main thread by a negative number. The *assignment_states* variable stores the actual state of the literal propagation, more precise the clause flags and the actual propagation literals. Variable *produced_globval* indicates if a new global assignment has been found in the last round for this worker.

```
typedef struct Worker Worker;
struct Worker {
        pthread_t thread;
        int id;
        AssignmentState assignment_state;
        int produced_globval;
};
Worker boss, *workers;
```

Listing 2: Worker Type used for Multithreading

A simple schema of the implementation can be found in Alg. 4, which also includes the optimization presented in the following section.

6.5.3 Cache Optimization

Sequential Assignments During the development of the algorithm it has been observed that the CPU's cache is not utilized very well. To overcome this an additional method has been introduced. The idea is to handle more than just one assignment vector when iterating over the clauses. The implementation is very straightforward: instead of using one single *assignment_state* as shown in List. 2 an array containing multiple assignment states is used.

Additionally the storage of the assignments has to be adopted to fulfil the new conditions. The base implementation has a very simple storage scheme for assignments: it is just an array which is indexed by the literal. This array therefore has as much elements as literals in the formula. By handling multiple assignments sequentially the number of elements is multiplied. The memory layout is chosen so that the locality of the sequential assignments is very high.

Figure 1 shows how the assignments storage is organized. It is arranged in three levels, with the thread at the highest level, followed by the literal and sequential level.

Regarding the algorithm's structure of nested loops this storage layout generates the most cache locality possible.

[7]POSIX Threads - a POSIX standard for developing multi-threaded application on POSIX compliant operating systems

Figure 1 also depicts the way assignments are initialized with the literals to propagate. The order of setting the propagation literals is as follows: $1, -1, 2, -2, ..., n, -n$. The number of parallel assignments that can be handled is 8 in this example (2 per vector $\times$ 2 sequentials $\times$ 2 threads). The consequence is that following literals can be propagated in the first round: $1, -1, 2, -2, 3, -3, 4, -4$.

A single assignment is described by 3 parameters: thread, sequential and entity number. When initializing an assignment, e.g. thread 0, sequential 0 and entity 1, and the literal to propagate is 1, the corresponding assignment vector is calculated (which is at address 0 in this case). The first entity is then set to 00, stating that literal 1 is assigned positively. To extend this example, if the assignment of thread 1, sequential 0, entity 2 should propagate -3, the second entity in the vector at address 16 is set to 11, which depicts a negative assignment.

The example shown in Fig. 1 assigns propagation literals like described in Table 13.

Figure 1: Assignment storage layout

```
+-----------+-----------------------------------------------+
|           |0                   1                   2      |
| array pos |0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3|
+-----------+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| bit     3 |1|0|0|1|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|1|0|0|0|0| \_second entity
|         2 |1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1| /
|         1 |0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0| \_first entity
|         0 |0|1|1|0|1|1|1|1|1|1|1|1|1|1|1|1|0|1|1|0|1|1|1|1| /
+-----------+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|thread     |0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1|
|literal    |1 1 2 2 3 3 4 4 5 5 6 6 1 1 2 2 3 3 4 4 5 5 6 6|
|sequential |0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1|
+-----------------------------------------------------------+

nr_threads = 2, nr_literals = 6, nr_sequentials = 2, nr_bits = 4

memory_pos = thread * nr_literals * nr_sequentials +
             (literal-1) * nr_sequentials +
             sequential
```

To give an idea of how parallelism and sequentials have been implemented, Alg. 4 shows a rough sketch of the required changes to the simple bit-parallel algorithm. The two most important sections are the following:

- the for all parallelization loop, where the *pthread* workers are spawned
- the *initialize next assignment* in line 8, which is a procedure protected by a semaphore – this procedure is the central building block of establishing work dispatching

Distributed Assignment Storage The previously introduced assignment storage in a single pre-allocated block of memory has a disadvantage. Whenever an assignment is

Table 13: Schema of assigning propagation literals to assignment vectors

propagation literal	assignment #	thread	sequential	entity
1	1	0	0	1
-1	2	0	0	2
2	3	0	1	1
-2	4	0	1	2
3	5	1	0	1
-3	6	1	0	2
4	7	1	1	1
-4	8	1	1	2

Algorithm 4 Parallelism and sequentials in bit-parallel algorithm

```
function BlBpParSeqFLP(formula)
    initialize global assignment
    repeat
        reset propagation literal list                    ▷ start a new propagation round
        for all workers in parallel do                    ▷ spawn threads
            while propagation literals left or changed do
                for all sequentials do                    ▷ initialize assignments for all sequentials
                    if not changed then
                        initialize next assignment        ▷ for details see Alg. 3
                for all clause in formula do
                    for all literal in clause do          ▷ detect unit clauses
                        for all sequentials do calculate clause states
                    for all literal in clause do          ▷ apply new units
                        for all sequentials do apply new units to assignments
                for all sequentials do                    ▷ handle failed literals of sequentials separately
                    if not changed then
                        if found failed literals then
                            handle failed literals
        synchronize workers
    until no new failed literals found                    ▷ terminate if no worker found
                                                          ▷ new failed literals
    return global assignment
end function
```

retrieved or needs to be set, a global offset is calculated. This offset calculation needs to include the literal, thread-id and the sequential-id.

As an optimization to get over this offset calculation the assignment storage can been moved to the worker object (which is used for parallelization issues). This also has the advantage to store data often used by a single thread closer together than before and therefore has better caching behaviour. This alternative assignments storage is refered to as *distributed storage* in this work.

The effects of both cache optimization techniques are discussed in the following experiments.

7 Experiments

This section contains experiments of the different variants of the developed failed literal probing algorithm. Every experiment tries to argue about the expectations and the outcome of the benchmarks.

A benchmark suite is created that is able to run the algorithm on arbitrary problems in every required configuration. The result of a run contains the following informations:

- *units*: the number of unit literals that have been found.
- *rounds*: the number of rounds that were necessary to finish (see also: rounds in Sect. 6.5.2).
- *iterations*: how often the whole formula has been iterated over.
- *assignments*: how many assignments were made through propagations.
- *vector utilization*: the ratio of how many bits are actively utilized in an iteration (the last iteration before a whole new assignment block is initialized is omitted – no bit is utilized in this case).

  The value is calculated like this: in the end of every iteration the number of active assignments is evaluated and summed up, this value is then divided by the total assignments that could have been active. This total value consists of the number of iterations where active assignments existed times the number of assignments per vector times the number of sequential assignment vectors.
- *runtime*: the real runtime[8] (wall-clock time).

All the benchmarks have been executed on a benchmark machine provided by the institute of formal models and verification (FMV) at JKU Linz. The features of this machine are as follows: 12 cores (two Intel® Xeon® DP E5645 6x2.4GHz), 96 GB main memory, Ubuntu-Server 11.10 Oneiric Ocelot (64 bit). The used compiler was GCC 4.6.1 (Ubuntu/Linaro 4.6.1-9ubuntu3).

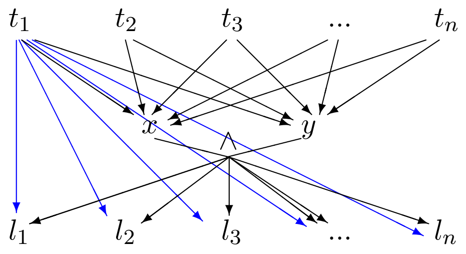
7.1 Problem Sets

Three problem sets with different properties have been developed to obtain reasonable results.

- **2sat** – A random SAT problem where every clause contains exactly two randomly selected literals with randomly selected polarity. These two literals are ensured not to be the same. One parameter is used when generating such a problem that indicates how many literals and clauses should be generated. Every clause triggers at least two propagations when running the algorithm.

[8]real runtime means the actual time that the program needs from start to termination; another way to measure time is the CPU time, which depicts the time the CPU is busy executing the program

Figure 2: Visual implication schema of quadratic problem set (implications from t_i to l_j are only shown for t_1)



- **quadratic** – A specific SAT problem that leads to a quadratic number of propagations while no failed literals are found. Its implications are designed as follows (also shown visually in Fig. 2). There are three different kinds of literals:
  - Two intermediate literals that build a connection between the other two kinds of literals. These two literals are denoted x and y in the example.
  - A set of triggering literals that basically imply both intermediate literals by two binary clauses. Every triggering literal is denoted by t_i where i is the number of literals used. It is important to say that a literal only occurs in one polarity because otherwise it would lead to a failed literal when propagating an intermediate literal. The example shows that for every triggering literal there exist exactly two clauses, namely one implying x and the other one y.
  - A set of implied literals which are implied by $x \wedge y$. Every implied literal is denoted by l_j where j is the number of implied literals used. Every l_j results in a clause $\neg x \vee \neg y \vee l_j$.

  The building block for a quadratic problem looks like the following, in the generated problems i is always equals to j.

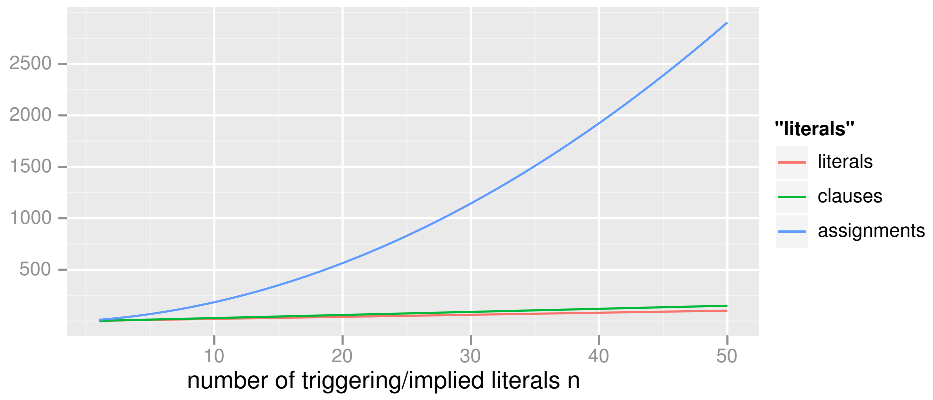
  $\neg t_i \vee x$
  $\neg t_i \vee y$
  $\neg x \vee \neg y \vee l_i$

  Figure 3 shows how many literals, clauses and assignments are required for problems with n triggering/implied literals (x-axis). The corresponding values can be calculated with these formulæ:

  Equation 11.
$$\begin{aligned} literals &= 2 * n + 2 \\ clauses &= 3 * n \\ assignments &= n^2 + 8 * n + 4 \end{aligned}$$

- **maxrounds** – A constructed SAT problem that leads all literals of one polarity to fail and therefore results in a CNF where all clauses have length one - every literal is fixed to a value. It is constructed equivalent to Ex. 7.

Figure 3: Characteristics of constructed quadratic problems.



Example 7. $\neg l_1 \vee l_2$

$\neg l_1 \vee \neg l_2$

$l_1 \vee \neg l_2 \vee l_3$

$l_1 \vee \neg l_2 \vee \neg l_3$

$l_1 \vee l_2 \vee \neg l_3 \vee l_4$

$l_1 \vee l_2 \vee \neg l_3 \vee \neg l_4$

...

This problem is specifically hard to solve for the developed algorithm, because it probes multiple literals at once in a fixed order. If probing $l_1/\neg l_1$ to $l_4/\neg l_4$ in parallel (8 assignments/16 bits), it happens that l_1 is recognised as failed literal and $\neg l_1$ is permanently assigned. As a consequence of permanently assigning $\neg l_1$ the assignment l_2 gets a failed literal. The problem is that both literals are watched at the same time so that a new round has to be started for l_2 to be identified a failed literal. This behaviour repeats until all failed literals are found.

In Sec. 7.3 an approach to reduce this negative effect is presented, which introduces randomness when selecting literals to probe.

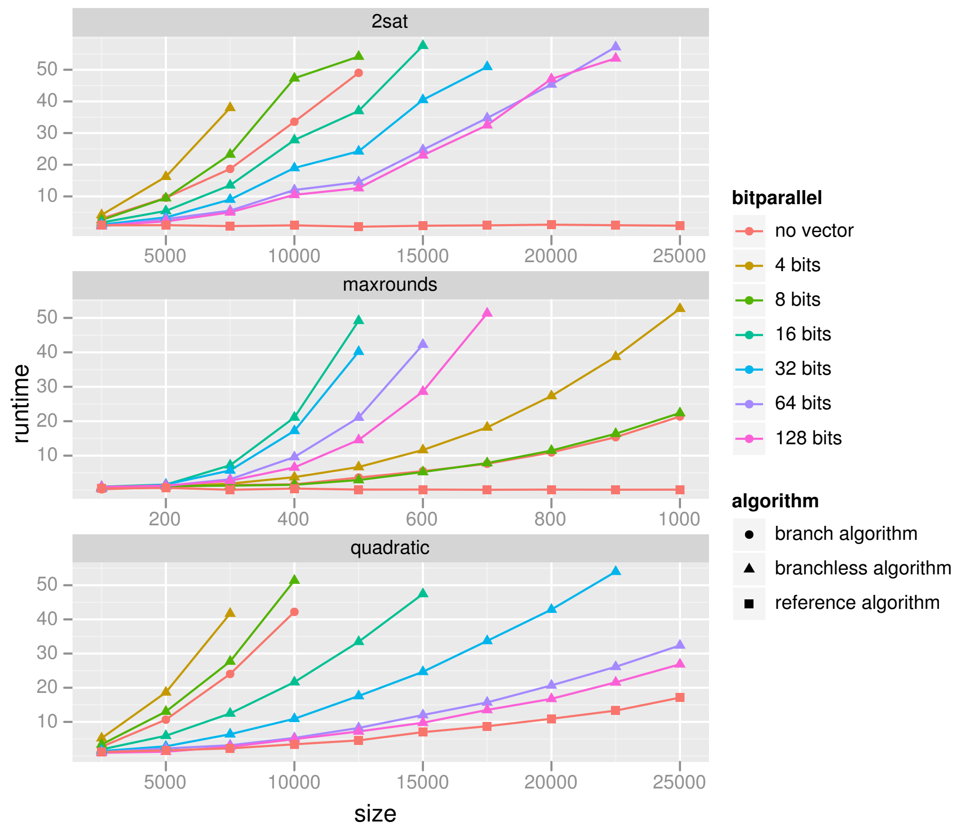
7.2 Efficiency of Bit-Parallelism

The major question regarding the introduction of bit-parallelism is how many bits must be utilized to overcome the overhead that is produced.

Implementing the algorithm in a branch-less manner introduced the need for a second iteration of a clause to set possibly found unit clauses. Another issue is that only the main part of the algorithm is branch-less and can be handled in parallel, the other parts still take as long or even longer than it was the case with the basic algorithm.

A very basic issue is that the required steps until propagation ends are very diverse for every literal. The problem is that a number of assignments are joined together, lets say 16, and start propagation. If there is just one assignment that takes a long time to finish propagation, and the other 15 terminate quite fast, those 15 assignments keep unchanged. These unused assignment 'slots' lead to a lowered vector utilization but handling assignments of a bit-vector separately would introduce large overhead which does not pay

Figure 4: Branch vs. branchless bit-parallel version of the algorithm with different sizes of bit-vectors used.



off. All these facts lead to the assumption that at least three or four assignments must be handled in parallel to be competitive to the basic version of the algorithm.

Figure 4 shows the performance of the algorithm with different vector lengths compared to the initially developed branching version. As predicted the overhead of the bit-parallel implementation just pays off somewhere between vector sizes of 8 to 16 bits. What also can be observed is that every doubling of the vector size almost speeds up linearly in the ideal case (as it is the case with quadratic problems).

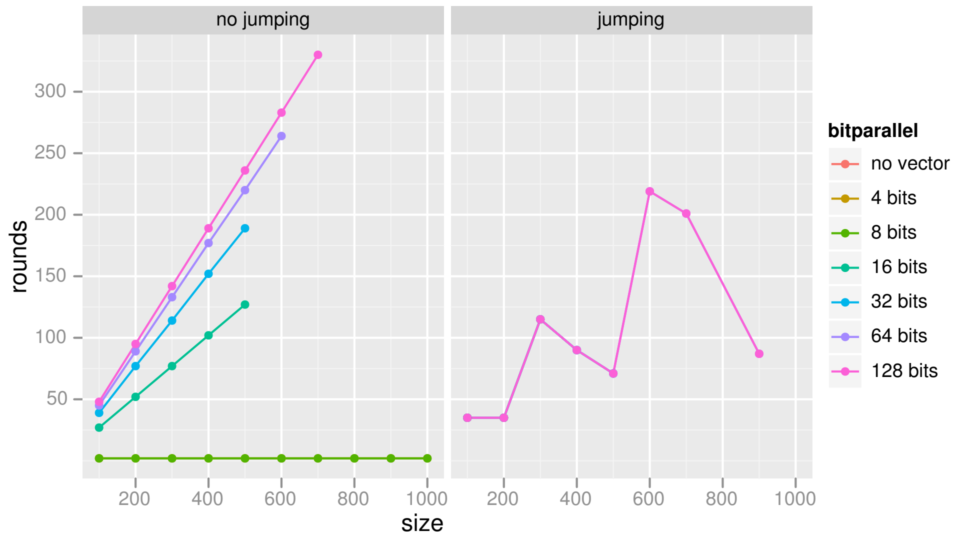
The usage of *SSE* unfortunately does not seem to scale that well because of additional implementation overhead - from 64 to 128 bits it is impossible to maintain linear speed-up.

The *maxrounds* result plot may seem odd for a moment. As already mentioned at the problem description before, this has to do with the implementation issue that the order of literal propagation is simply ascending from 1 to n. This results in many rounds and therefore bad runtimes. This does not hold for bitvector sizes with 4 or 8 bits as depicted in the plot.

7.3 Next Literal Jumping

As mentioned before, the *maxround* problem causes the algorithm to do many rounds. This is because of the ascending order of literals to probe. By introducing *literal jumping*

Figure 5: Comparison of rounds when employing literal jumping on maxrounds problems



it is possible to reduce this effect. Literal jumping is implemented like this:

1. a random literal is determined as the start literal
2. a step distance is calculated, it must be a value relatively prime to the number of literals (that is to find a number which greatest common divisor to the number of literals is one)
3. beginning from the start literal you can add the step distance to find the next literal to propagate, ensuring that every literal is exactly once covered - until the start literal is calculated again

Figure 5 shows how required rounds are influenced by literal jumping. If bit vectors are longer than 16 bits its influence is positive - otherwise more rounds are necessary. The same effect would be generated if the source CNF file is scrambled up in a way that literals are arbitrarily exchanged.

7.4 Multi-Threading

The upper bound of speed-up achieved through multi-threading in this algorithm seems to be sub-linear. One limiting factor is that the initialization of a new propagation round for a worker is a global operation and therefore needs locking over all threads. This is also the case when a new failed literal has been found and it is added to the global assignment.

Another fact is that the more literals are handled in parallel the less information gained in previous iterations can be utilized in further ones. The most limiting factor for gaining speed-up with multi-threading is, that unit propagation generally is a sequential task and in worst case cannot be parallelized at all (see Sect. 3.4).

Figure 6: Comparison of single- and multithreaded algorithm with up to 32 threads

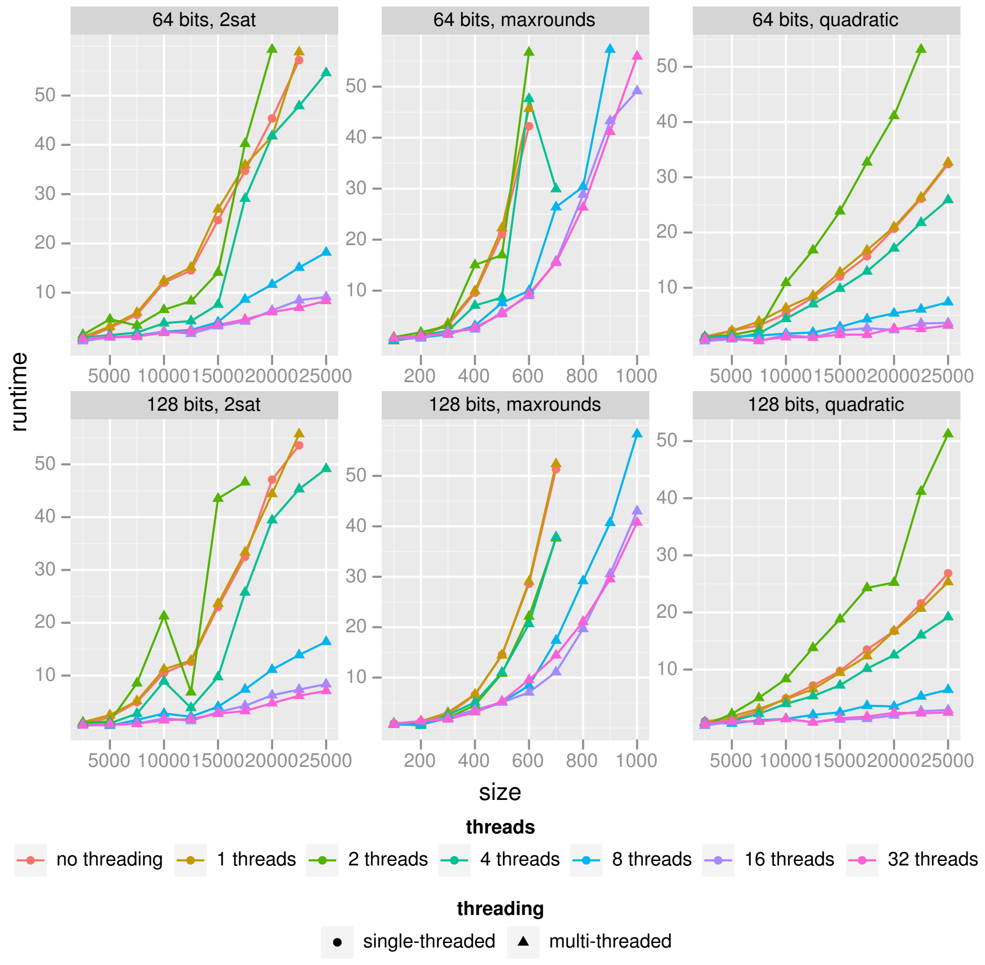


Figure 6 shows how multi-threading influences performance on different problems with vector sizes of 64 and 128 bits. There are various inferences that can be made from these plots:

- if comparing the no threading (no locking) version to the one thread version they almost produce same results; whereas the two thread version performs really bad for an unknown reason.
- *2sat* and *quadratic* problems seem to benefit most of employing multiple threads, speed-up seems to be limited at approximately 8, when utilizing 32 threads - which is a nice value for a 12-core machine.
- the maxrounds problem also allows speed-up by multi-threading since rounds can be processed faster.

7.5 Sequential Assignments

As already mentioned in Sect. 6.5.3, the use of multiple assignment states can help to utilize the CPUs cache better. Since the algorithm has to fetch an assignment for a specific literal from a more or less arbitrary position in memory anyway, this fact can be exploited to fetch multiple assignments. A reasonable speed-up is expected if utilizing two or more of those sequential assignments, depending on the size of the CPU cache and global memory bandwidth utilization.

As there can be seen in Fig. 7 speed-ups of almost 40 percent are possible, depending on the problem class and size. It is clear that the *maxrounds* problem does not perform better due to the structure it belongs to. The *2sat* and *quadratic* problem really show the intended behaviour and show that caches can utilized better than before.

Subsequent experiments with combinations of multi-threading *and* sequential assignments will show that they heavily influence each other and one has to find an optimal configuration to perform best.

7.6 Distributed Assignments

This improvement heavily influences performance in multi-threaded benchmarks but also shows moderate improvements in single-threaded results - which can be attributed to reduction in offset calculations.
The performance plots in Fig. 8 show how distributed storage influences the results of the three problem classes with different numbers of used threads. It is obvious that the CPU cache works more efficient if assignment storage is local for each thread.

When looking at the speed-up compared to the assignment block version, distributed assignments has better scalability. Even with maxround problems a considerable gain in performance can be achieved.

When looking at the result for four threads a strange behaviour can be observed. The memory block version performs really bad whereas the distributed storage implementation

Figure 7: Performance effects of employing sequential assignments in the algorithm

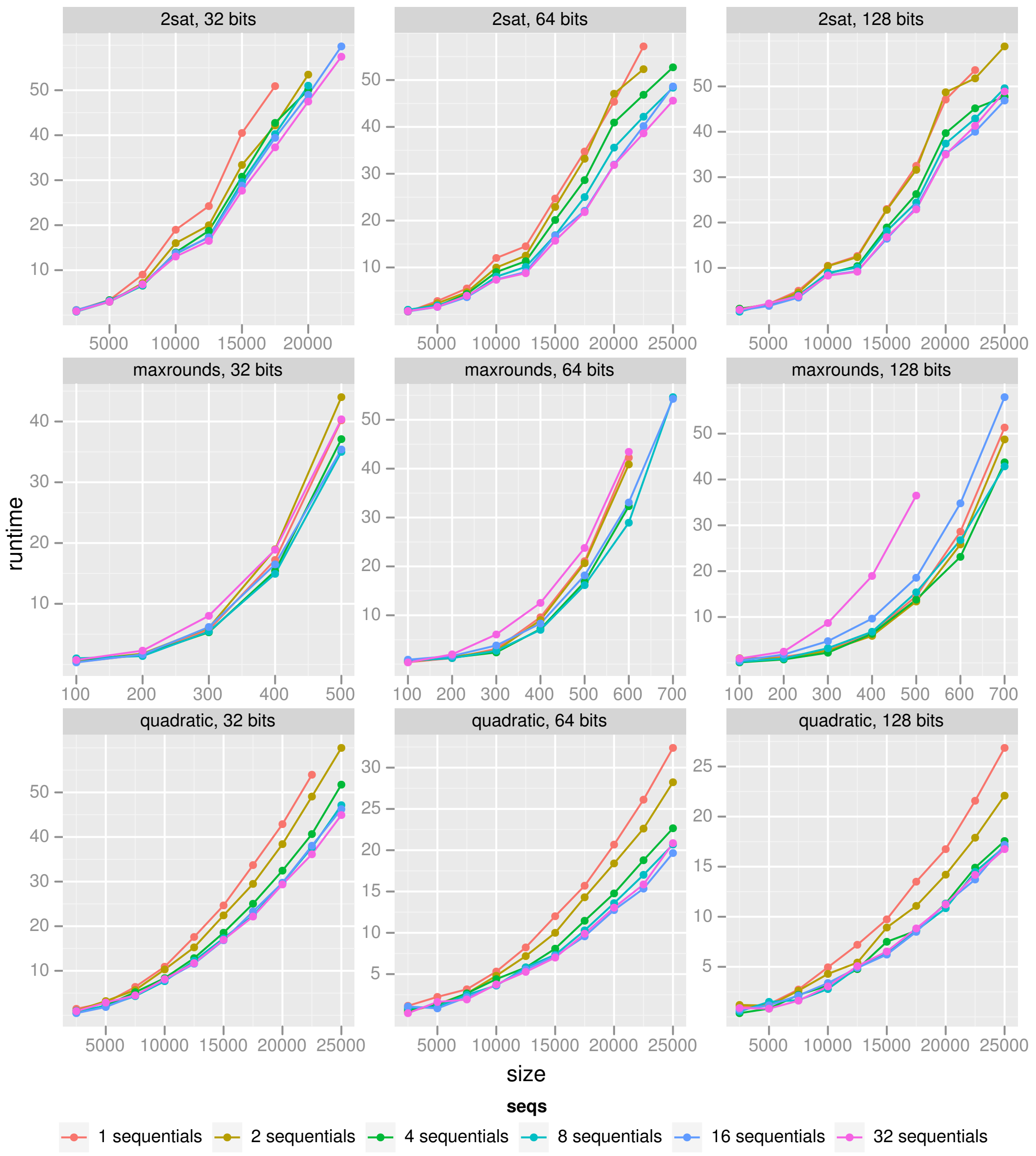
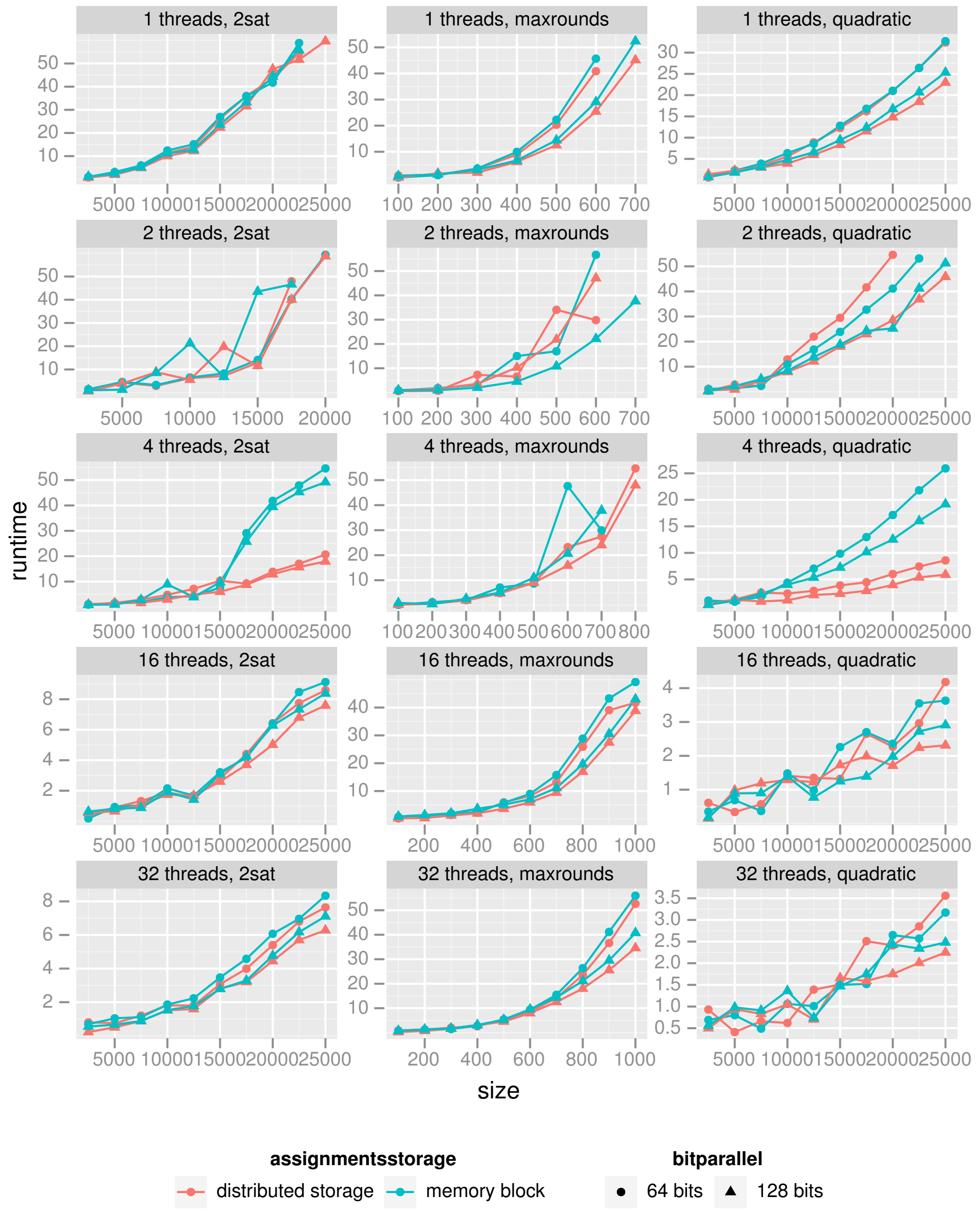


Figure 8: Comparison of different assignment storage approaches with different problem classes and threading options



scales almost linearly with the problem size. The cause for this has to be a caching effect, no other explicable reason can be found.

What can also be seen is that two threads perform really bad for both storage solutions. The *quadratic* problem even has a negative speed-up, which also has to be a mystifying problem with CPU caching.

Another observation made is that the distributed storage technique has a more positive effect on the 128 bit version of the algorithm when looking at the *quadratic* problem. The 64 bit version even runs slower than the memory block version.

7.7 Bit-Vector Utilization

A very interesting measurement on this algorithm concerns utilization of bit-vectors. This value gives an idea of how much time is spent in calculations that really yield results. The unused bits can be seen as a lost resource - the calculations have to be done anyway until nothing changes in the whole vector.

Figure 9 shows the vector utilization for the three problem classes and a diverse number of sequential assignments to see how this influences the utilization value. The *2sat* problem shows that smaller vector sizes equals higher utilization which is the expected behaviour for this problem class. It can also be observed that even 32 sequential assignments do not reduce utilization - but what is interesting is the poor value for 128 bit vectors. The average is about 0.15 and 0.2 which means that with an optimal utilization of 1.0 a speed-up of approximately 5 would be possible.

The plots for the *maxrounds* problem give an idea, why longer bit-vectors result in bad performance. Compared to the 4- and 8-bit version the longer ones have utilizations that go towards zero.

Since the *quadratic* problem is constructed to have a well-defined length of implication chain only small variance occurs. Nevertheless with dynamic generation of new assignments there is a great potential to speed up propagation.

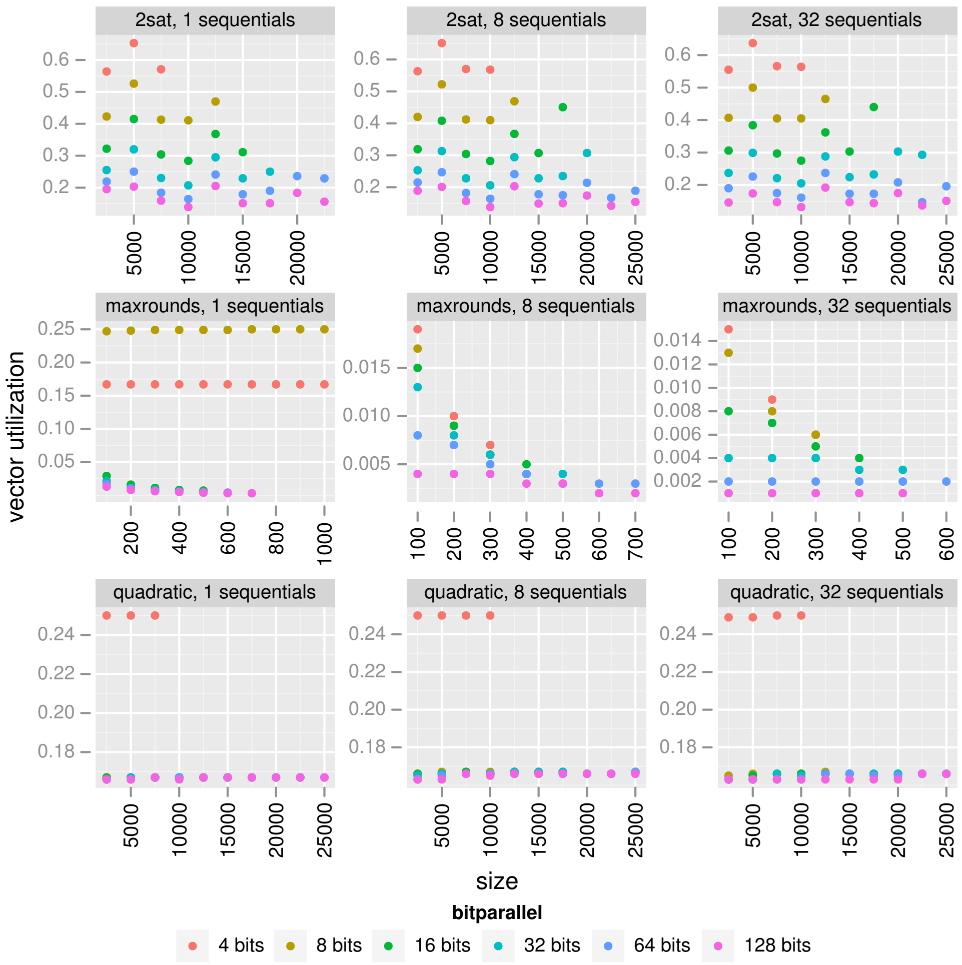
The poor utilization values of 128 bit vectors have to be reviewed from a different angle. Since no additional hardware (SSE is available in almost every modern CPU) is needed to be able to handle long vectors less utilization can be neglected. As the other evaluations reveal, these bit vectors perform best while at the same time no additional costs arise.

In general it would be a substantial improvement to be able to populate the vector with new work efficiently. The problem is that the initialization of the assignment vector is a complex task. No way has been found in this work to establish this in a reasonable amount of time (e.g. in branch-less manner).

7.8 Optimal Configuration

This experiment has been used to determine which configuration yields best total results in terms of minimal runtime. We already observed that distributed storage of assignments outperforms the memory block implementation, so this decision is already fixed.

Figure 9: Vector utilization values for different vector lengths on different problem classes



What has to be noted is that this results only apply to the specific benchmark machine. The performance is heavily influenced by machine specific properties like number of cores, memory bandwidth and cache sizes.

A substantial unanswered question is which combination of threads and sequentials has to be chosen to perform best. Figure 10 shows the result of the benchmark. Note that the problem size has been fixed for this plot (*2sat* and *quadratic* have problem size 25000; *maxrounds* has problem size 500).

It can be seen that there is a point in the *2sat* problem where additional sequential assignments do not affect the performance positively whereas adding more threads still reduces the runtime. The turning point is reached with 8 threads. When using less than 8 threads good reductions in runtime can be achieved with *2sat* and *quadratic* problems. As through all benchmarks, the *maxrounds* problem does not perform well with sequential assignments, but with multiple threads.

When looking at the overall best reachable runtimes it is observed that employing 32 threads combined with two sequential assignments achieves best results. The best single-threaded configuration is using 32 sequential assignments. What holds for both configurations is that bit-vectors with a size of 128 bits perform best - barely better than the 64 bit version.

A comparison of the optimal configurations to the reference algorithm *sflprepc* by Armin Biere is shown in Fig. 11. Table 14 shows the configurations of these three programs. The problem configurations are the same as before. First of all it can be seen that the speed-up of the parallel version is really good: 8.2 for *2sat*, 3.5 for *maxrounds* and 8.4 for *quadratic* problem set. Since the benchmark machine is equipped with two hexacore CPUs resulting in 12 cores total this gives a positive prospect of what can be possible with much more cores.

Table 14: Optimal program configurations

bar in figure	algorithm	threads	sequentials	vector size	assignment storage
1	branch-less	1	32	128	distributed
2	branch-less	32	2	128	distributed
3	sflprepc	1	-	-	-

It is no surprise that the reference algorithm performs better with *2sat* and *maxrounds* problems, since it needs much less iterations due to occurrence lists. An idea of the potential of the developed algorithm is shown with the *quadratic* problem: in this case many propagations have to be done and occurrence lists are not that helpful. The challenges therefore really fit the strength of the developed algorithm. Even the single-threaded version terminates faster than the reference algorithm, the multi-threaded version is about nine times faster.

Figure 10: Runtimes when varying number of threads and sequentials to find optimum (applied to fixed problem size)

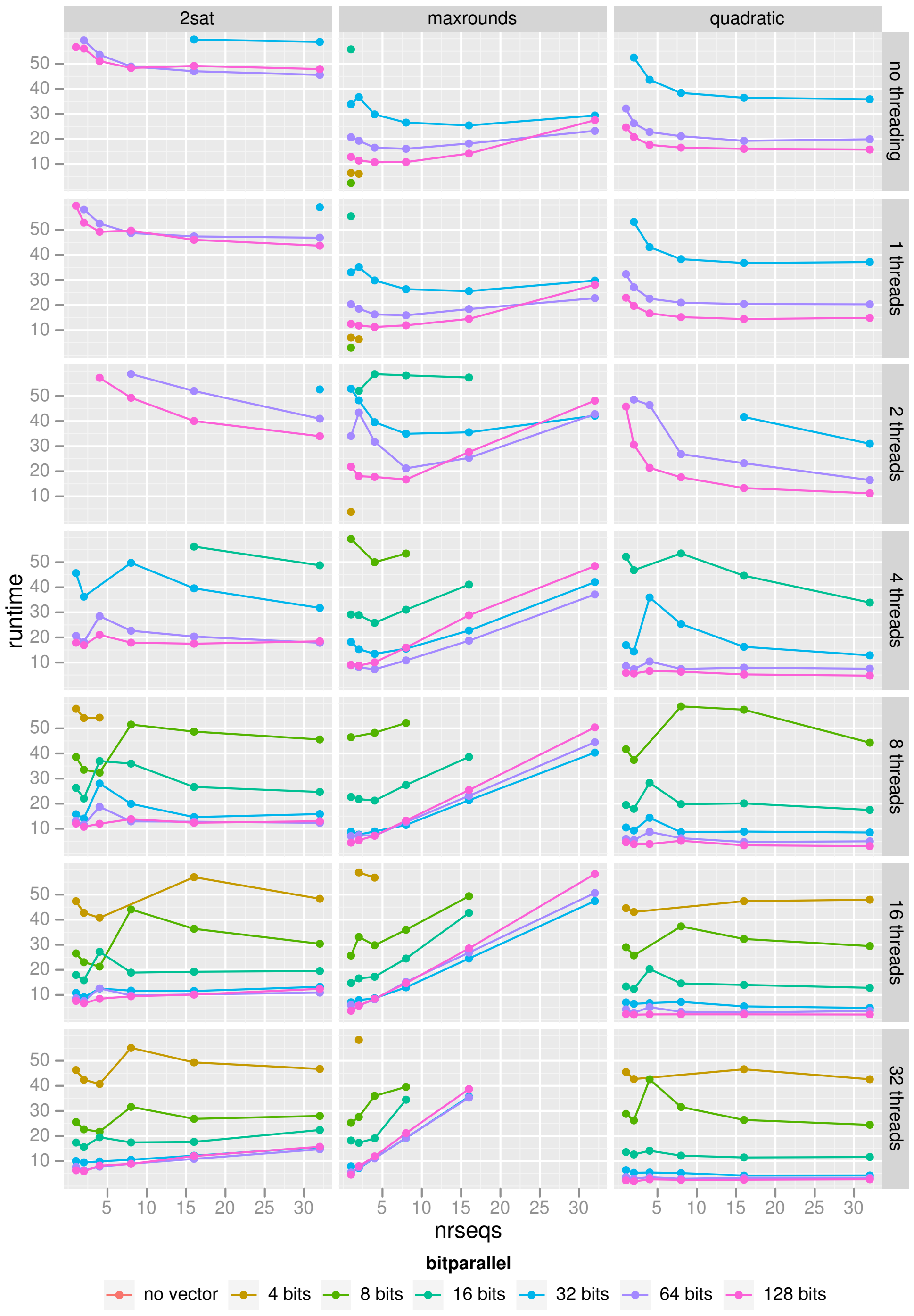
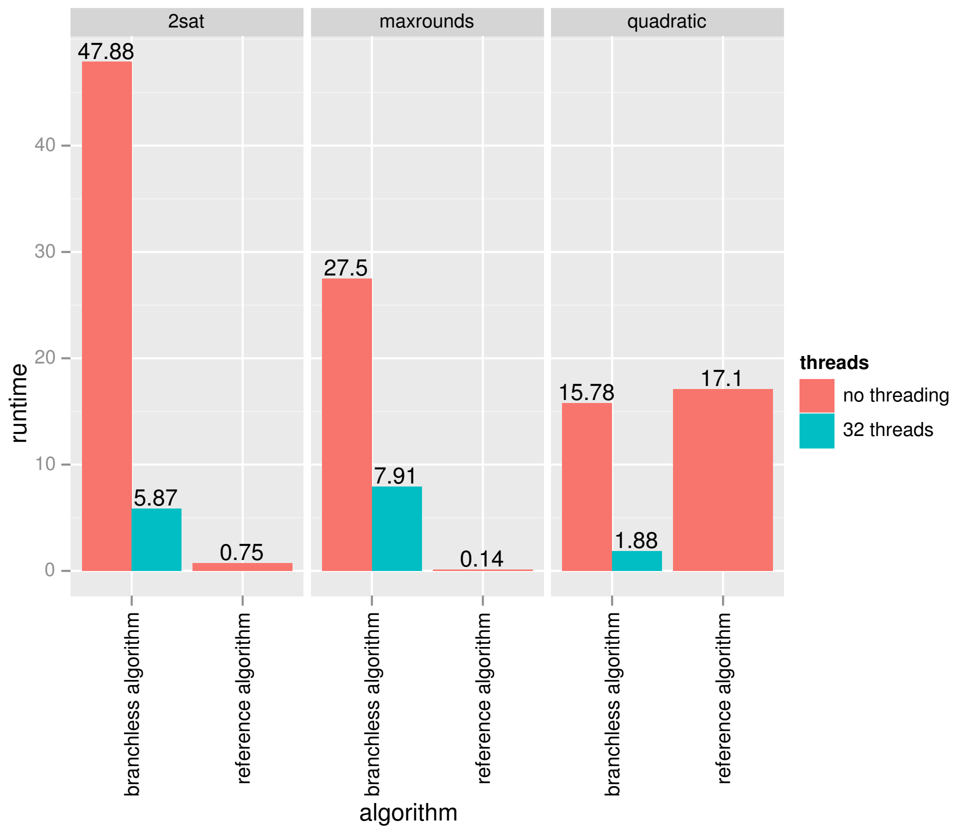


Figure 11: Comparison of runtimes of optimal configurations with reference algorithm



The evaluation can be summarized to these core points:

- utilization of long bit-vectors scale well until additional overhead has to be introduced (compare native 64 bit-vectors and SSE2 128 bit-vectors).
- multi-threading scales really well with streaming-oriented algorithms like the developed algorithm (at least until memory bandwidth limits are reached).
- CPU caches have to be utilized well to obtain good results in such memory intense algorithms by localizing memory accesses (using sequential assignments or optimized storage structures).
- long bit-vectors lead to lower bit-vector utilization if not repopulating them on the fly - this fact gives a significant potential to speed-up the developed algorithm in future versions.
- the developed algorithm complies to the design policies of well-performing GPGPU algorithms and therefore is expected to perform well on such massively parallel architectures.

8 Conclusion

This thesis gave an overview on SAT solving in general, focused on complete SAT solving algorithms that are based on *DP*. This is because the central component of *failed literal probing* and those algorithms is identical, namely *boolean constraint propagation*.

In Sect. 3 the current state-of-the-art in parallel SAT solving has been discussed to have an idea of what is the difference to this thesis' objective. It also shows that the general interest on parallelism in SAT solving is as high as never before in SAT solving history.

Theoretical background on failed literal probing has been presented in Sect. 4, which includes the state-of-the-art in *boolean constraint propagation* as well as possible reasoning techniques.

Related work on bit-parallelism was discussed in Sect. 5. A work by Heule and van Maaren chased a similar approach like this thesis, but, contrary to this work, in the domain of incomplete SAT solvers.

Section 6 covered the implementation of the different stages of the algorithm - starting from the basic algorithm with no optimizations to the final implementation. The utilized data structures as well as complex bit-operations were explained in detail, including various pseudocode which revealed the algorithms behaviour. The implemented optimizations were described in further subsections.

The most important part of this thesis, namely the experiments and evaluations were presented in Sect. 7. As the evaluation summary in the end of Sect. 7.8 depicts, the developed algorithm revealed various valuable facts.

However, the target of this thesis never was to implement an algorithm that outperforms state-of-the-art *BCP* implementations. It should evaluate the potential of 'streaming' algorithms applied to this kind of application field.

8.1 Future Work

As already mentioned in the motivation in Sect. 1.1, this thesis is a step into the direction of porting the developed algorithm to massively parallel systems like GPGPUs. This thesis should be extended to get an implementation that will utilize the whole parallelism of a GPU in the long term view.

What has to be done first is to design a schema how parallel threads are organized: the number of dimensions, blocks per dimension and threads per block. The next step is to decide which parts of the algorithm should be executed on the GPU and which other parts to process on the CPU. The last step is to implement the kernels that are executed on the GPU, where every kernel must avoid branch divergence to perform well.

It is hard to predict if the GPU implementation would outperform state-of-the-art implementations since caching behaviour is unknown for this special implementation. It is also unclear how processor frequency and memory bandwidth influences the performance.

Nevertheless there is also optimization potential for the developed algorithm, e.g. extending the use of bit-parallelism throughout the algorithm, increasing bit-vector efficiency or improving assignment storage and operations like proposed in Sect. 6.4.2.

The other way of future work concerns the implementation of additional probing techniques like lifting, hyper-binary resolution or equivalent literal detection, which can be found in Sect. 4. Especially lifting can be implemented in an easy way due to the structure of the algorithm.

Anyway, this thesis should be a good basis for future work on streaming implementations of *boolean constraint propagation*. This work has revealed that notable speed-ups can be achieved when applying parallelism to the algorithm, which give an optimistic perspective.

References

[1] H. M. Adorf and M. D. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Neural Networks*, pages 17–21, San Diego, Calif., 1990.

[2] Christoph M. Wintersteiger Antti E. J. Hyvarinen. Approaches for multi-core propagation in clause learning satisfiability solvers. Technical report, Microsoft Research, UK, 2012.

[3] Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *Eighteenth national conference on Artificial intelligence*, pages 613–619, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[4] Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.

[5] Armin Biere. The evolution from limmat to nanosat. Technical report, Dept. of Computer Science, ETH, 2004.

[6] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Comput.*, 29(7):969–994, July 2003.

[7] Max Böhm and Ewald Speckenmeyer. A fast parallel sat-solver - efficient workload balancing. In *Annals of Mathematics and Artificial Intelligence*, pages 40–0, 1996.

[8] Marco Cadoli and Andrea Schaerf. Compiling problem specifications into sat. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, pages 387–401, London, UK, UK, 2001. Springer-Verlag.

[9] Geoffrey Chu, Peter J. Stuckey, and Aaron Harwood. Pminisat - a parallelization of minisat 2.0. Technical report, Department of Computer Science and Software Engineering, University of Melbourne, 2008.

[10] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[11] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in satisfiability problems. In *In Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27, 1993.

[12] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)*, AAAI'94, pages 1092–1097, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[13] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[14] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[15] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[16] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[17] Jon William Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, Philadelphia, PA, USA, 1995. UMI Order No. GAX95-32175.

[18] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (sat) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1996.

[19] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel sat solving. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 499–504, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[20] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *JSAT*, 6(4):245–262, 2009.

[21] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Efficient cnf simplification based on binary implication graphs. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 201–215, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] Marijn J.H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Accepted for HVC 2011*, 2012. Accepted for HVC 2011.

[23] Marijn J.H. Heule and Hans van Maaren. Parallel sat solving using bit-level operations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:99–116, 2008.

[24] Marijn J.H. Heule and Hans van Maaren. *Look-Ahead Based SAT Solvers*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 5, pages 155–184. IOS Press, Amsterdam, handbook of satisfiability edition, February 2009.

[25] MJH Heule. March, towards a lookahead sat solver for general purposes. Master's thesis, Masters thesis, TU Delft, The Netherlands, 2004.

[26] Edward A. Hirsch and Arist Kojevnikov. Unitwalk: A new sat solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1-4):91–111, January 2005.

[27] Henry A. Kautz, Ashish Sabharwal, and Bart Selman. Incomplete algorithms. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 185–203. IOS Press, 2009.

[28] Matthew Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pages 926–931, Washington, DC, USA, 2007. IEEE Computer Society.

[29] Inês Lynce and João Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '03, pages 105–, Washington, DC, USA, 2003. IEEE Computer Society.

[30] Norbert Manthey. Parallel sat solving—using more cores. In *Pragmatics of SAT*, 2011.

[31] Ruben Martins, Vasco Manquinho, and Ines Lynce. Improving search space splitting for parallel sat solving. In *Proceedings of the 2010 22nd IEEE International Conference on Tools with Artificial Intelligence - Volume 01*, ICTAI '10, pages 336–343, Washington, DC, USA, 2010. IEEE Computer Society.

[32] Ruben Martins, Vasco Manquinho, and Ins Lynce. An overview of parallel sat solving. *Constraints*, 17:304–347, 2012.

[33] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a sat problem. *J. Autom. Reason.*, 24(1-2):165–203, February 2000.

[34] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the eighth National conference on Artificial intelligence - Volume 1*, AAAI'90, pages 17–24. AAAI Press, 1990.

[35] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[36] T. Schubert, M. Lewis, and B. Becker. PaMiraXT: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222, 2009.

[37] João P. Marques Silva and Karem A. Sakallah. Grasp– a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[38] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *J. Artif. Int. Res.*, 32(1):565–606, June 2008.

[39] Ramin Zabih. A rearrangement search strategy for determining propositional satisfiability. In *in Proceedings of the National Conference on Artificial Intelligence*, pages 155–160, 1988.

[40] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4-6):543–560, June 1996.

[41] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.

[42] Ying Zhao, Sharad Malik, Matthew Moskewicz, and Conor Madigan. Accelerating boolean satisfiability through application specific processing. In *In Proceedings of the International Symposium on System Synthesis (ISSS), IEEE*, pages 244–249. Princeton, 2001.

Lebenslauf

Persönliche Daten

Robert Aistleitner
Maierhof 8, Tür 2
4283 Bad Zell

Tel.: +43 (0)664 783 03 35
E-Mail: robert.aistleitner@gmail.com

Geboren am 18. Mai 1987 in Linz
Familienstand: ledig
Staatsbürgerschaft: Österreich

Bildung

seit 11/2010	Masterstudium Informatik an der JKU Linz
10/2007–11/2010	Bachelorstudium Informatik an der JKU Linz
09/2001–06/2006	HTBLA Perg - EDV und Organisation
09/1997–07/2001	Hauptschule Tragwein
09/1993–07/1997	Volksschule Allerheiligen im Mühlkreis

Beruf

seit 07/2009	Software-Entwickler bei La Gentz KG
07–09/2008	Ferialjob Software-Entwickler bei AMS Engineering GmbH
02–09/2007	Software-Entwickler bei Engel Austria GmbH
08/2006–01/2007	Präsenzdienst
2002–2005	Diverse Berufspraktika

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Bad Zell, am 10. Februar 2013

Robert Aistleitner, BSc