ZetaSAT – Boolean SATisfiability Solving in a Desktop Grid Environment

Wolfgang Westje Wolfgang Blochinger Wolfgang Küchlin
Symbolic Computation Group
University of Tübingen
Sand 14, D-72076 Tübingen, Germany
http://www-sr.informatik.uni-tuebingen.de
{westje,blochinger,kuechlin}@informatik.uni-tuebingen.de

Sebastian Wedeniwski
IBM Business Consulting Services, Germany
wedeniws@de.ibm.com

Abstract

This paper presents ZetaSAT, a parallel Boolean satisfiability (SAT) solver that is specifically designed for deployment in Desktop Grid environments. ZetaSAT is based on the Grid platform ZetaGrid. Our work particularly addresses specific issues arising in the execution of constraint satisfaction problems of the kind of SAT in Desktop Grids, like problem decomposition, scheduling, and termination detection. We report on performance measurements indicating the usefulness of our approach.

1 Introduction

Desktop Grid Computing [24] aims at harnessing idle resources of desktop computer systems for collaboratively treating resource intensive problems. Typically, the number of employed compute nodes ranges from hundreds (e.g. workstations of corporate networks) to many thousands (Internet scale Distributed Computing). Since current end-user desktop computer technology features immense CPU power, for appropriate applications, even small-scale Desktop Grids are capable to deliver supercomputer-level performance at extremely low costs.

In this paper we report on the deployment of a prominent constraint satisfaction problem (CSP) in a Desktop Grid environment using the ZetaGrid [34] platform. To the best of our knowledge, this work represents the first research effort on employing Desktop Grid technology for tackling hard constraint satisfaction problems.

The specific CSP we are considering is Boolean satisfiability (SAT) Solving. The SAT problem asks, whether there exists for a given Boolean formula F a variable assignment such that F evaluates to TRUE. SAT was the first problem shown to be NP complete [11]. Besides this theoretical relevance, SAT lies at the core of many important applications from diverse domains, like electronic design automation (EDA) [3, 33, 31], artificial intelligence [23], scheduling [12], and cryptography [26].

There still exist unsolved SAT encoded problems in all major application fields. Particularly the ever increasing complexity of chip designs is a source of extremely hard SAT problem instances which are far too complex to be solved by employing state-of-the-art SAT solvers on current sequential or conventional parallel hardware.

In order to efficiently exploit the massive parallelism of Desktop Grids, applications must exhibit specific characteristics, most importantly a considerable degree of parallelism and scalability. Due to the highly volatile environment, limited network performance and the possibility of off-line computation, a potential application must also permit problem decomposition into loosely coupled parallel tasks to be farmed out for computation.

For the SAT problem both criteria are fulfilled. But in stark contrast to all other prominent Desktop Grid applications, problem decomposition has to be carried out in a fully dynamical manner. To break down exponential runtime complexity barriers, modern SAT solving algorithms rely on sophisticated heuristics which can reduce the computation time for specific problem instances. Employing appropriate heuristics can speed up the computation time by several orders of magnitude. It is clear though, that the benefit of a heuristics can not be predicted for a particular problem instance and varies considerably. This holds for the whole problem, as well as for individual subproblems to be treated in parallel. Consequently, it is virtually impossible

to predict the run-time of an individual subproblem in advance, making dynamic problem decomposition inevitable.

These properties of SAT are also intrinsic to other constraint satisfaction problems and represent a significant challenge for deploying a parallel CSP application in a Desktop Grid environment which exhibits the aforementioned limitations.

By enabling the important class of CSPs for Desktop Grid execution, this paper contributes to the understanding on which classes of problems can be tackled using Desktop Grid technology. Our work demonstrates that not only embarrassingly parallel problems can be solved on Desktop Grids and thus extends the scope of Desktop Grid technology. Specifically, we make the following contributions:

- We show that parallel SAT solving (as a prototypical example for a constraint satisfaction problem) can be beneficially carried out in Desktop Grids.
- We realized a high performance parallel SAT solver that has the potential to push the limit of solvable SAT instances.
- We demonstrate that the well-known ZetaGrid platform serves as a general-purpose platform for deploying applications on Desktop Grids.

The rest of our paper is organized as follows. In Section 2 we discuss related work. Section 3 gives a brief account of the SAT problem and modern SAT solving techniques. Section 4 provides an overview of the ZetaGrid platform. In Section 5 we give a detailed description of our ZetaSAT system. We report on performance measurements in Section 6. In Section 7 we conclude and give some directions for future research.

2 Related Work

2.1 Desktop Grid Applications and Platforms

In this section we discuss other major (computational) applications that have been deployed on Desktop Grids and some prominent Desktop Grid platforms which share similar characteristics to our ZetaGrid platform.

Maybe the most popular Desktop Grid project is *SETI@home* [28, 32], which aims at finding extraterrestrial life by analyzing data delivered from radio telescopes. Each compute client gets a work unit of fixed size (about 340 KB) representing 107 seconds of data for analyzing.

The project *distributed.net* is primarily concerned with carrying out brute-force cracking of cryptographic code (RC5-56, RC5-64, RC5-72, DES, and CSC projects). Equal sized blocks of keys from the whole key space are farmed out for testing on the compute nodes until the right key is found.

The goal of the *GIMPS* (Great Internet Mersenne Prime Search) project [19] is to find mersenne prime numbers. These are prime numbers of the form $2^P - 1$. Depending on the capability of a compute node it gets one of three kinds of tasks. Powerful nodes carry out a first time primality test, mid-class clients do double checking and slow clients are assigned with factoring out small exponents. First time primality tests require several days of compute time even on the most powerful clients.

Simulation of protein folding kinetics is studied by the *Folding@home* project [15, 29]. Each compute client runs a 100 psec molecular dynamics simulation, calculates the energy variance for this time period, and returns this data to the server.

The sibling project *Genome@home* [17] is concerned with the design of protein sequences. Here, compute clients receive a set of protein backbone coordinates and design parameters from the server and execute a protein design algorithm. Upon completion, the protein structure is communicated to the server and stored for postprocessing.

A characteristic which all the aforementioned Desktop Grid applications share, is that they carry out a semi-static problem decomposition. Essentially, this means that individual subtasks need not be decomposed further once their computation has been started. Here, parallelism and scalability can easily be achieved by adapting the problem size of the parallel tasks before execution. This is in contrast to the requirements we face when parallelizing constraint satisfaction problems like SAT. As stated above, for parallel SAT solving, it is inevitable to carry out a fully dynamic task decomposition process in order to achieve an appropriate degree of parallelism and scalability for execution on a Desktop Grid.

The commercial *Entropia* Desktop Grid platform [9] has been used for realizing a wide range of applications, including parameter studies, monte carlo applications and combinatorial testing (e.g. sequence analysis and virtual drug screening). The open source project XtremWeb aims at providing a light weight infrastructure for carrying out research on Desktop Grid, Global Computing, and Peer to Peer distributed systems.

Entropia, XtremWeb, and ZetaGrid basically implement the same architecture and programming models. While ZetaGrid and XtremWeb are open source and highly platform independent, Entropia only supports Microsoft Windows based platforms and no source code is disclosed.

2.2 Parallel SAT Solving

Böhm and Speckenmeyer proposed a parallel SAT solver for a Transputer system composed of 256 processors [8].

This work mainly studies efficient load balancing techniques for parallel SAT solving of hard randomly generated SAT instances.

Zhang's PSATO [35] is a distributed parallel SAT solver for networks of workstations. PSATO is based on an external parallelization approach of the sequential prover SATO. This work concentrates mainly on solving open quasigroup existence problems.

The parallel SAT solver PaSAT by Blochinger *et al.* [6] is focussed on establishing an efficient distributed parallel learning process between the nodes of a cluster or a network of workstations. Also cross-fertilization of different kinds of heuristics is addressed [7]. PaSAT has been successfully employed in an industrial application [5]. It is based on the parallel platform DOTS [4].

All discussed parallel SAT solvers are designed for tightly coupled parallel hardware and are not intended to be deployed in Desktop Grid environments.

GridSAT [10] is a parallel SAT solver targeted for Globus Grids. It employs grid technology to make reservations of appropriate dedicated resources (like compute clusters) for executing a parallel SAT solver. In contrast, ZetaSAT addresses the specific challenges of parallel SAT solving in non-dedicated Desktop Grid environments, like limited resources, user initiated resource preemption, and failure of compute nodes.

3 The SAT Problem

3.1 Basic Definitions

The Boolean satisfiability (SAT) problem consists of determining for a Boolean formula F a variable assignment such that F evaluates to TRUE or to prove that for F no satisfying variable assignment exists. Typically, F is specified in conjunctive normal form (CNF), which is a representation into which all Boolean formulae can be easily transformed. In CNF, a formula is composed of conjunctions ($\wedge$) of *clauses*. A clause is the injunction ($\vee$) of one or more *literals*, and a literal is a variable or the complement of a variable. Consider the following Boolean formula in CNF:

$$F = (x_1 \vee x_3) \wedge (x_2 \vee \overbrace{\overline{x_3}}^{literal}) \wedge (\underbrace{\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}}_{clause}) \wedge x_3$$

The variable assignment $x_1 \rightarrow$ FALSE, $x_2 \rightarrow$ TRUE, $x_3 \rightarrow$ TRUE represents a satisfying assignment of F.

A fundamental property of a formula in CNF is that it is satisfiable iff. in each clause at least one literal evaluates to TRUE. If in a clause all but one literals have already been assigned to FALSE the remaining literal must be assigned with TRUE in order to satisfy the clause. Such clauses are called *unit clauses*. A situation when all literals of a clause of F are assigned to FALSE is called a *conflict*, the corresponding clause is called a *conflicting clause*.

3.2 Modern SAT Solving Algorithms

The classical Davis-Putnam-Logemann-Loveland (DPLL) SAT solving algorithm [14, 13] was proposed in the early 1960s. While still representing the algorithmic framework of modern complete SAT solvers, it has been significantly enhanced by introducing advanced heuristics for pruning the search space of variable assignments to be considered. Most beneficial advances could be achieved by employing *dynamic learning* and *conflict driven backtracking* techniques [25, 2]. It turns out that these enhancements are specifically effective for real world problem instances. Moreover, by applying sophisticated implementation techniques significant performance improvements could be achieved [37].

Figure 1 shows the top-level structure of an enhanced version of the DPLL algorithm that employs modern dynamic learning and conflict driven backtracking techniques. The algorithm is given in the iterative layout used by the sequential zChaff implementation [27], which forms the basis of our parallel version (see Section 5). We will restrict our discussion to a top-level treatment. For a more detailed explanation, the reader is referred to the literature (for example, see [37]).

Basically, the DPLL algorithm belongs to the class of backtracking search algorithms. During this process, partial variable assignments are speculatively extended in order to find a satisfying assignment. The procedure `decide()` decides according to a heuristics [21] which unassigned variable should be chosen next for extending the current partial variable assignment. It also decides on the phase of the chosen variable. We call this process a *decision*. Each decision is recorded on an *assignment stack* along with an associated *decision level*. The decision level of the first decision is 1 (we will discuss the role of decision level 0 later).

The purpose of the procedure `deduce()` is to infer additional variable assignments that are necessary for the formula to be satisfied considering the current partial variable assignment. After making a new decision, some clauses may have become unit clauses. The `deduce()` procedure assigns the corresponding variables such that the unit clauses become TRUE. Variable assignments forced by unit clauses are called *implications*. Implications are also recorded on the assignment stack at the current decision level. The iteration of the above procedure is called *unit propagation*. Unit propagation stops when either no unit clauses exist any more or a conflict occurs. In the first case a new decision is made inducing the next decision level. In

```
while(true) {
  if (decide()) {                                    // Decision
    while(deduce() == CONFLICT) {                    // Deducing
      if (current_decision_level == 0) {    // Top-Level Conflict
        return UNSATISFIABLE;
      } else {
        new_decision_level = analyze_conflicts();    // Learning
        back_track(new_decision_level);            //Backtracking
      }
    }
  } else {                              // no unassigned variables
    return SATISFIABLE;
  }
}
```

Figure 1. The DPLL Algorithm with Conflict Driven Backtracking

the second case the procedure `analyze_conflicts()` is invoked which constitutes the core of modern SAT solvers. It performs two important tasks:

- **Dynamic Learning**: A new clause called *conflict clause* or *lemma* is constructed by analyzing the reasons for the current conflict. The lemma is added to the input formula representing an additional constraint that prevents the occurrence of the same conflict in other regions of the search space.
- **Conflict Driven Backtracking**: The conflict is resolved by undoing variable assignments (of potentially several decision levels).

Conflict analysis employs a so called *implication graph* which basically depicts the history of all current unit propagations by recording implication relationships. The implication graph is a directed acyclic graph (DAG). Its vertices represent variable assignments (decision or implication assignments). Incoming edges of a vertex indicate the reasons for the assignment, i.e. assignments of variables in the corresponding unit clause. Note that vertices corresponding to decision assignments don't have incoming edges. A conflict is indicated by two conflicting vertices (representing opposite assignments of a variable). Conflict analysis is carried out by bi-partitioning the implication graph into a *conflict side* and a *reason side*. The conflict side contains at least the two conflicting vertices and the reason side contains at least all vertices corresponding to decision assignments. There are several valid schemes for performing such a bi-partition of the implication graph, we refer the reader to [36] for a detailed description. The set of vertices on the reason side which have an edge into the conflict side comprise the reason for the current conflict, in other words, the corresponding set of variable assignments $R = \{L_1, \ldots, L_n\}$ (expressed as set of literals L_i) represents a sufficient condition for the considered conflict to occur. By adding the lemma $\ell = \overline{L}_1 \vee \ldots \vee \overline{L}_n$ to the input formula we can prevent that the search process repeats the same combination of variable assignments in the further search. This avoids that the same conflict is regenerated in another region of the search space, reducing the total search space to be treated. Since the underlying construction mechanism of lemmas is resolution it can easily be shown that adding lemmas does not affect the correctness of the algorithm.

The backtracking level is computed as the lowest decision level at which exactly one literal of the lemma is unbound. Note that initially the lemma is a conflicting clause, and that by construction, the number of unbound variables of a lemma cannot move from zero to more than one when releasing all variable assignments of one decision level.

All variable assignments recorded on the assignment stack up to the computed backtracking level are released by the procedure `back_track()`, thus resolving the current conflict. The newly added lemma, which is now a unit clause, takes the search to a new direction.

A special case arises, when the backtracking process reaches decision level 0. In this case, the current lemma forces a variable assignment at level 0, called a *top-level assignment*. Backtracking to level 0 means, that the corresponding conflict cannot be resolved by releasing any non top-level assignments. Thus, top-level assignments have been proven to be a necessary condition for the input formula to be satisfiable. Consequently, top-level assignments are fixed during the further search process.

If all variables have been assigned without the occurrence of a conflict the input formula is satisfiable. Unsatisfiability of the input formula is proven by the occurrence of a *top-level conflict*, i.e. a conflict at decision level 0. A top-level conflict implies unsatisfiability, since it cannot

be resolved by releasing variable assignments (top-level assignments are fixed).

Typically, the overall solving process is steered by a *solver strategy*. A solver strategy determines for example policies for performing *clause deletion* and *restarts*. Extensively adding lemmas to the input formula can slow down the deduction process and thus potentially outweigh the performance improvements of dynamic learning. A solution to this problem is to periodically delete lemmas selected according to a heuristics. Search restarts are performed in order to prevent the search process from getting stuck in a futile part of the search space. The search process is periodically cancelled by backtracking to level 1 and restarted keeping some results (typically lemmas) of the previous run. A crucial point when realizing restarts together with clause deletion is to preserve completeness [1]. A solver is called *complete* if it eventually terminates either delivering a solution or indicating unsatisfiability. One mean to preserve completeness of a solver that performs restarts and clause deletion is to gradually increase the restart interval.

4 The ZetaGrid Platform

ZetaGrid was initially developed and tested in the IBM-Lab Böblingen, Germany. It is a successful example of Desktop Grid Computing, involving about 10,000 computers in a heterogeneous and dynamic environment, i.e. the IBM intranet and the Internet. It was designed to solve problems which are very large and need lots of CPU power. The first project of ZetaGrid was a mathematical one – the verification of the zeros of the zeta function. That's where the name comes from.

ZetaGrid is an open source and platform-independent computational grid system. The goal of ZetaGrid is to use free capacity of voluntarily participating computers. Therefore, it schedules a configured large number of loosely coupled and independent work units of different static sizes. Every computer with free capacity can request those work units – in size or number corresponding to a user-defined capacity. The participator may complete the work unit offline or online. The completed work units are then delivered back to the server, where the results are verified. To make verification of results possible, the work units contain overlapping information and can be redistributed or recomputed. Successful and regular participators are rewarded with trust factors, so that they will be treated preferentially, i.e. that their results do not have to be checked rigorously and that they can obtain more work units.

4.1 Architecture

There are two main actor groups in a desktop-grid environment: the *job-submitters* who want problems to be solved and the *resource providers*, that allow access to their resources through the Desktop Grid platform. ZetaGrid is logically organized in three layers to reflect the needs of these actor groups (following the terms by Germain et al. in [18]):

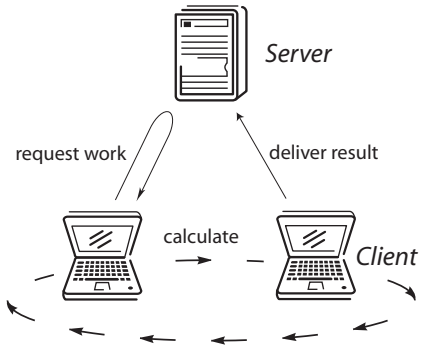

Figure 2. ZetaGrid's Communication Cycle

- through the *request layer* job submitters can upload problems, setup parameters and retrieve results
- the *broker layer* decomposes the problem, schedules and merges subproblems, detects the end of the solving process and collects statistics
- within the *service layer* subproblems are solved. This is access point for the resource providers.

ZetaGrid is a platform specialized in a dynamic environment with a large number of participating computers. The operational model of the platform is a three-tier architecture consisting of web server, the application server, and the database. This setup is known to be scalable for high-performance computing tasks [16]. It also allows to have a differently strict security policies for different tiers and makes administration tasks easy because of its centralized nature.

The rich client is implemented in Java and communicates only with the web server of the middle tier via stateless sessions using HTTP. In this concept, a client connects the server to synchronize the client version, to request work units, and to deliver completed work units. The server never connects a client to get any data. The transfer of work to the clients and the result communication back to the server runs along the communication cycle of *request work unit* $\rightarrow$ *calculate* $\rightarrow$ *deliver result* as shown in Figure 2. This cycle ends if the client is stopped (e.g. by terminating the screen saver) or if there is no work available. In that case the client periodically polls for new work units.

All interaction is initiated and defined by the participating user, since the user is the only person who can judge the free capacity of their resources. He can choose to run the client as a low-priority background process or in

screensaver mode. ZetaGrid is optimized for all kinds and speeds of connections, e.g. modem, LAN, WAN.

One of the most important factors for the acceptance of such a computational grid system is to ensure privacy for the participating clients. ZetaGrid provides a secure Java kernel which secures its communications and activities by restricted layer access with digital signatures and key establishment protocols. Furthermore, it makes every source (except private keys) open for the community. The client is entirely written in Java so that Java2 Security concepts, i.e. the Java Sandbox, can be enabled to ensure that an untrusted- and possibly malicious-application cannot gain access to system resources.

5 ZetaSAT

ZetaSAT is a research effort to bring efficient parallel SAT solving into the Desktop Grid. By designing the SAT solver with this difficult environment in mind, several solutions for problems posed through the Desktop Grid have been found.

ZetaSAT is centrally organized in a client-server-manner. The clients solve subproblems and asynchronously deliver results to the server. Additionally they periodically report a certificate of the progress of their search. The server keeps backlog of the communication and schedules the subproblems. It also executes problem decomposition when necessary, checks results and detects whether a solution to the whole problem has been found.

5.1 Overview

Developing a parallel SAT solving application in a Desktop Grid environment poses several challenges, some of which are:

- the parallel algorithm has to cope with
  - volatile resources
  - limited storage/network capacity
  - result tampering
- the Desktop Grid platform has to support
  - fully dynamic problem decomposition
  - finite applications demanding a timely solution

We tackle these problems in the following way:

The volatility of resources is taken into account by dynamic scheduling and fully dynamic problem decomposition (see Sections 5.3.2 and 5.3.1). Newly added resources are given subproblems that are split off of other subproblems. Resources that leave the Grid are detected via a method that is similar to the heartbeat method: When clients do not contact the server in a certain time frame, their part of the search space is rescheduled to another machine. So it is made sure the whole search space is covered. This also helps significantly in finding a timely solution to a problem, as the lost subspace is recovered early. Other applications, like ZetaGrid's zero verification or the key cracking in distributed.net do an off-line check for lost work units, which is possible because of their semi-static way of problem decomposition.

Limitations in storage and network capacity are addressed in Section 5.3.4. Only short certificates of the search progress ('top-level assignments', cf. Section 3) are communicated to lighten the load on the network. The big part of the solver progress, the accumulated lemmas, are periodically saved locally, where also a selection heuristics is applied to cope with limited storage space.

A common problem with publicly deployed Desktop Grids like SETI@home or distributed.net is that users (resource providers) deliver results that have been tampered with. In ZetaSAT this problem is attacked by strongly checking results, that claim that the formula is satisfiable. The given model is applied to the formula and only if this test succeeds, the problem is marked solved. Results other than SAT are not yet verified, but strategies like double checking of results are possible, though not as easy as with other applications, due to the non-static nature of problem decomposition.

When surveying Desktop Grid platforms to run a parallel SAT solver on, we focussed on robustness and extensibility. Extensibility was especially necessary to allow for further research on our non-typical application. We chose ZetaGrid as our platform mainly because of its maturity (running for about 3 years, costantly updated[1]) and the fact that the sources for clients and servers are freely available, which facilitates the adjustment of the platform for our application.

Similar reasoning lead us to the SAT solver we employ: zChaff by L. Zhang, which combines being open source and delivering high performance (it has been winner of the 2002 SAT competition in the category "Best performance of a complete solver on industrial benchmarks") [30].

5.2 Architecture

ZetaSAT inherits the core of ZetaGrid's architecture. Most importantly the communication cycle illustrated in Figure 2 is also the way of the communicating in ZetaSAT. The logical three-tier-architecture is adopted as well, but in

[1] see http://www.zetagrid.net/zeta/news.html for ZetaGrid's update history

a different manifestation. In ZetaSAT the request layer is realized via a web interface to allow job-submitters to comfortably upload formulas and follow the solution progress of their problems. For management purposes also main parameters of the system, like the size of the subproblem pool (cf. Section 5.3.2), can be adjusted by a web page. ZetaGrid's broker layer had to be extended as to allow fully dynamic problem decomposition (cf. Section 5.3.1). In the meantime, our extensions have been incorporated into the latest release of ZetaGrid. ZetaSAT's service layer, i.e. the client portion of our system, has been realized on top of the ZetaGrid client, using the SAT solver zChaff in a dynamic library that is accessed from the Java-based client via the Java Native Interface. This library is downloaded by each client whenever there is a new version. As already stated in Section 4, it is cryptographically signed. A difference between ZetaGrid's default mode of operation and ZetaSAT's is that the encryption of results when transferred from client to server is disabled. The reason for this is, that ZetaGrid's way of encryption considerably slowed down the result communication of the SAT application. We are surveying other means of secure communication to make this feasible in a future release.

5.3 Parallel SAT solving with ZetaSAT

Improving sequential SAT solvers has been a very active field of research in recent years, mainly driven by the need of hardware designers to verify their designs. One goal of our distributed SAT solver was to transfer the progress made in sequential SAT solving to parallel SAT solving.

For parallelizing an algorithm one has to answer several questions: How can the problem be decomposed that enough parallelism is yielded? How is the scheduling of subproblems to processors done, in order to best use the available resources? How are subproblems merged and the end of the computation detected?

ZetaSAT's approach to problem decomposition is the fully dynamic problem decomposition explained in Section 5.3.1. Fully dynamic problem decomposition requires that parallel tasks which are already executed on a compute node, can be decomposed further into subtasks in order to generate enough parallelism for available idle processors. Since it is impossible to predict the number of subtasks that are actually generated during a computation, additionally a termination detection mechanism must be employed, which is elaborated in Section 5.3.3.

Both, dynamic task decomposition and termination detection can be realized by a central or by a distributed approach. Implementing a distributed approach requires peer-to-peer functionality of the underlying Desktop Grid platform. Since this is not the case with ZetaGrid (as well as with all other major Desktop Grid platforms) we have chosen to pursue a centralized strategy.

In Section 5.3.2, the needs and mechanisms of scheduling in a Desktop Grid environment are elaborated. We conclude our description of ZetaSAT with an explanation of the checkpointing mechanisms employed in ZetaSAT.

5.3.1 Problem Decomposition

In typical Desktop Grid applications like the ones described in Section 2, subproblems sent out to clients have no further relevance for the problem decomposition.

The opposite is true for ZetaSAT's problem decomposition, there is a strong dependency between a result of a client and the following subproblems. So it is crucial that none of the results gets lost. How ZetaSAT manages lost work units is explained in Section 5.3.4.

The key to understanding the problem decomposition is the notion of top-level assignments as explained in Section 3. These are a necessary condition for further search. In sequential solvers, backtracking to level 0 or preprocessing of the formula adds top-level assignments[2].

The parallel algorithm splits the search space into two disjoint subspaces by heuristically choosing a variable and adding two complementary assignments for this variable to the top-level of the respective subspaces. These assignments are the assumptions for the subproblems.

Because both search spaces are disjoint, they can be searched in parallel. The result of the formula can simply be determined as follows: In the case of a satisfiable subproblem, the whole formula is satisfiable. If both subproblems are unsatisfiable, then the whole problem is unsatisfiable, as both contrary assumptions lead to a conflict.

Further decomposition can be achieved by splitting subspaces, which leads to a tree of independent subproblems that can be solved in parallel. This method (theoretically) yields exponential parallelism.

It is clear that this parallel algorithm is complete and will eventually terminate, because with each split, the search space is getting smaller. Regarding performance though, it is not favorable to split the search space, since the powerful solver techniques described in section 3 are only employed once the solving process is sequential.

As a heuristics for choosing a good splitting variable we simply follow the choice of the underlying sequential solver, by using its first decision (i.e. the decision on level 1). The rationale behind this is, that in modern SAT solvers highly sophisticated decision strategies are implemented and research on parallel splitting heuristics is still to be done.

[2] literals occurring in only one phase, i.e. only negative or only affirmative are an example for top-level assignments through preprocessing

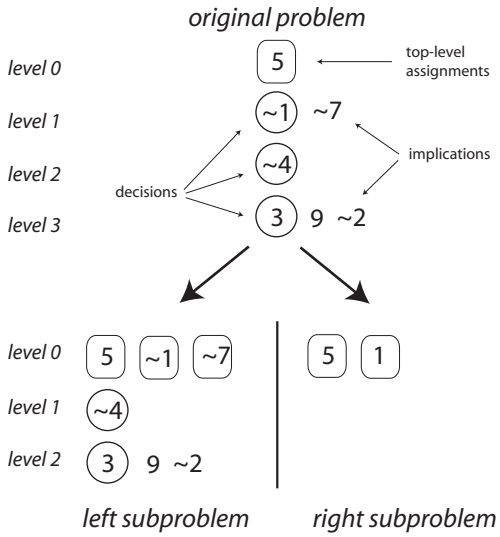

Figure 3. Assignment Stack Transformation

Because of the restriction imposed by the volatility of participating computers, we had to implement this problem decomposition theme in ZetaSAT in two ways: a server split and a client split. While the client split is optimized for performance and only works in the same environment as the original problem, the server split is less efficient, but works on any client.

The client split transforms zChaff's assignment stack in the same way as described in [10]. Figure 3 shows an example for the splitting process. The left subproblem is constructed by appending the whole first level of the assignment stack (i.e. the decision and all its implications) to the top-level assignments (level 0) and moving the other levels down one step. The right subproblem consists of a new level 0 with the appended complementary assignment of the decision variable of former level 1.

This transformation is efficient for this client, because the solver does not have to re-construct the assignment stack, but can continue from where it was interrupted for the split.

In case of a client leaving the Desktop Grid (maybe because the client computer was turned off), its assigned part of the search space has to be processed by another client in order to get to a timely solution. This client most probably does not have the same set of clauses, because it searched elsewhere for a solution or it just joined the grid. Then the client split is not possible as the implying clauses for the assignments on the right side of the assignment stack might not be in the new client's database.

This is the situation when a server split is made. The server creates the left subproblem by adding the decision assignment from level 1 (split variable) to the old level 0. The right subproblem is the same as in the client split, namely the old level 0 concatenated with the negated split variable assignment. Both subproblems only differ in a complementary last assignment on level 0. Using the example assignment stack from figure 3, the two server splits would be: $x_5, \bar{x}_1$ on the left and x_5, x_1 on the right.

The server split might be a loss of efficiency if executed within the same clause database, on the other hand it is also very similar to a successful technique called *restarts* where periodically all assignments except for the top-level ones are discarded (see Section 3).

When a split is necessary to yield enough parallelism, ZetaSAT performs a client split. Only in case of a client leaving the grid, the server split method is used (cf. Section 5.3.4).

5.3.2 Scheduling

Run times of SAT solvers on subproblems are unpredictable and so is the number of participants of the grid. This leads to the need of a dynamic scheduling to achieve the necessary degree of parallelism. In current parallel SAT solvers, most often the work-stealing technique is used [6, 10]. The work-stealing scheme operates as follows: if a worker becomes idle, it interrupts another worker and forces to split-off part of its search-space. This scheme requires the coordinator to contact a worker. Communication in ZetaSAT is strictly pull-based, so workers (i.e. clients) cannot be contacted by the coordinator.

In ZetaSAT we adapt this scheme by letting clients periodically report to the server. The server decides whether they have to split off part of their search space or not. To bridge the gap between servicing an incoming work request and the periodic reports, we use a subproblem pool. In the PSATO system by Zhang et al. a similar approach is taken when using a list of guiding paths [35].

As noted in Section 5.2, communication runs along ZetaGrid's communication cycle (see Figure 2). Every client periodically stops its solving process and sends a certificate (its level 0 of the assignment stack and the first decision assignment) to the server, which puts it into the subproblem pool. Then the client requests further work units. If the pool of subproblems is full, the server answers 'CONTINUE' and removes the subproblem from the pool, else it answers 'SPLIT' to let the client perform a split of the search space. In case of a split, the server keeps one half of the search space in the pool of subproblems.

There are also clients that did not report the state of their search, because of the fact that their search space is exhaustively searched or because they just joined the grid. Work unit requests of those clients are serviced with subproblems from the pool.

To bootstrap the solving process, the first work unit request is answered with the whole problem, i.e. with an

empty assignment stack.

In addition to relying on a pool of subproblems to keep all clients busy, ZetaSAT additionally implements feature to run several formulas at once. Whenever the pool of one formula is empty, client request are serviced through the other formula's pool. By the use of job-submitter controllable priorities for each formula, the job-submitter can decide for which formula a timely solution is more important. [3]

5.3.3 Termination Detection

As we employ fully dynamic problem decomposition, it is not clear beforehand how many subproblems will be distributed. So we have to check the results of the subproblems in order to detect the termination of the search. If the result of a subproblem is SAT then the client will also send a model which is checked by the server. If the check succeeds, the search is terminated. In ZetaSAT termination is detected in two ways: by looking for a result that is SAT and by counting pending subproblems. Pending subproblems represent open search spaces. Whenever the search space is split the counter of pending subproblems is incremented and whenever a result is delivered it is decremented. The counter is initialized to 1 when the whole problem is sent out. If the counter reaches the value 0, the search space has been exhaustively searched.

5.3.4 Fault Tolerance and Checkpointing

In ZetaSAT local and remote checkpointing is employed to react to the needs of a Desktop Grid environment. In every iteration of the communication cycle the current level 0 of the assignment stack as well as the first assignment on level 1 are transmitted to the server (i.e the certificate). The server records this subproblem in its database. The amount of data sent over the network is comparatively small (less than the number of variables of the formula), so one can do it rather frequently.

The client's clause database also contains valuable information for a fast solution process, as the lemmas generated through previous conflicts are stored there. The size of the database, however, might be exponential in the number of variables. ZetaSAT's approach is to periodically save part of this database locally.

Choosing the right lemmas for a checkpoint is a difficult task, because one cannot foresee the effect of a lemma in accelerating the search. A similar problem is tackled in clause deletion in sequential SAT solvers. Modern solvers delete clauses in order to speed up the unit propagation (see Section 3). A common way to choose the clauses to delete, is to do it by the value of its literals. A clause containing literals with the value TRUE cannot help pruning the search space. Clauses with few literals with unknown values and many literals that are set to FALSE might be a good source to get an early conflict and thus prune the search space. As this might work well with clause deletion, it does not help with writing a good checkpoint, as the assignment of variables at the time of writing the checkpoint might be different from the assignment when the clauses are read again. In ZetaSAT we decide which clause to write and which to discard only upon the maximum length of the clause, since short clauses have the potential to prune a bigger part of the search space than longer clauses.

The data saved through big local checkpoints is used again when clients reenter the grid and startup their client program (or if the screensaver starts running). Remote checkpoints are accessed whenever a client does not connect the server for a certain period of time. It is assumed then, that the client left the grid, so its part of the search space is given away to one or more clients, depending on the filling of the subproblem pool.

When designing the fault tolerance mechanisms in ZetaSAT, we had the choice of doing a lazy recovery of a lost work unit (i.e. only redistribute the subproblem when the rest of the search space has already been scanned for a solution) or a timely recovery (by the use of a local Time-To-Live stamp for every sent out subproblem). We took the latter approach, because it is closer to our splitting heuristics (we do a split according to the first decision of the sequential solver). We assume that earlier subspaces (i.e. subproblems that are split off earlier) have a bigger potential of incorporating the problem's solution than later ones.

6 Performance Measurements

To verify the usefulness of our Desktop Grid approach to parallel SAT solving, we measured ZetaSAT's peformance on a variety of benchmarks. The environment for our experiments is that of a typical research institution or small enterprise. We used 37 computers that acted as clients (see Table 1) and an Intel Xeon 2,6 GHz computer with 2GB main memory as server. The computers were connected by a 100Mbps Fast Ethernet network. Except for the Intel nodes the computer were shared with other users, as they were part of two workstation pools for students. ZetaSAT supports platform independence, so we could use three different operating systems for the experiments.

The sequential run times were measured on an 2,6GHz node using the current version (2003.12.04) of zChaff[4]. No modifications from the original setup were made. The zChaff library used for the parallel SAT solving was also run with the same default parameters.

[3] In the experiments this mode was disabled for fair results on single formulas.

[4] Available at http://ee.princeton.edu/~chaff/zchaff.php

Name	Result	Time Seq. (sec)	Time Par. (seq)	ZetaSAT Speed-Up
pyhala-braun-unsat-35-4-01.shuffled.cnf	UNSAT	1910	145	**13.17**
pyhala-braun-unsat-35-4-02.shuffled.cnf	UNSAT	734	149	**4.94**
pyhala-braun-unsat-35-4-03.shuffled.cnf	UNSAT	763	127	**6.01**
pyhala-braun-unsat-35-4-04.shuffled.cnf	UNSAT	671	134	**5.00**
pyhala-braun-unsat-40-4-01.shuffled.cnf	UNSAT	5563	326	**17.06**
pyhala-braun-unsat-40-4-02.shuffled.cnf	UNSAT	8295	292	**28.41**
pyhala-braun-unsat-40-4-03.shuffled.cnf	UNSAT	17664	381	**46.40**
pyhala-braun-unsat-40-4-04.shuffled.cnf	UNSAT	12548	306	**40.96**
pyhala-braun-sat-35-4-01.shuffled.cnf	SAT	374	84	**4.43**
pyhala-braun-sat-35-4-02.shuffled.cnf	SAT	529	84	**6.32**
pyhala-braun-sat-35-4-03.shuffled.cnf	SAT	169	89	**1.89**
pyhala-braun-sat-35-4-04.shuffled.cnf	SAT	420	89	**4.70**
pyhala-braun-sat-40-4-01.shuffled.cnf	SAT	928	136	**6.81**
pyhala-braun-sat-40-4-02.shuffled.cnf	SAT	2334	208	**11.22**
pyhala-braun-sat-40-4-03.shuffled.cnf	SAT	696	122	**5.72**
pyhala-braun-sat-40-4-04.shuffled.cnf	SAT	545	206	**2.64**
19_rule/SAT_dat.k50.cnf	SAT	943	251	**3.75**
19_rule/SAT_dat.k55.cnf	SAT	690	268	**2.57**
19_rule/SAT_dat.k60.cnf	SAT	1126	782	**1.44**
19_rule/SAT_dat.k65.cnf	SAT	2538	1097	**2.31**
19_rule/SAT_dat.k70.cnf	SAT	5838	1152	**5.07**
19_rule/SAT_dat.k75.cnf	SAT	9898	4709	**2.10**
19_rule/SAT_dat.k80.cnf	SAT	6071	2857	**2.12**
19_rule/SAT_dat.k85.cnf	SAT	10015	4410	**2.27**
19_rule/SAT_dat.k90.cnf	SAT	10903	6960	**1.57**
19_rule/SAT_dat.k95.cnf	SAT	43468	5898	**7.37**

Table 2. Experimental results for IBM hardware verification benchmarks and the Pyhala-suite

Name	Result	Time Seq. (sec)	Time Par. (seq)	ZetaSAT Speed-Up	GridSAT Speed-Up
6pipe.shuffled-as.sat03-414.cnf	UNSAT	1302	MEM OUT	**–**	1.23
Urquhart-s3-b1.shuffled.cnf	UNSAT	456	89	**5.14**	1.01
avg-checker-5-34.shuffled.cnf	UNSAT	1090	693	**1.57**	1.10
bart15.shuffled.cnf**(*)**	SAT	14934	59	**251.70**	8.18
cache-05.shuffled.cnf	SAT	167	147	**1.14**	1.11
cnt09.shuffled.cnf	SAT	5070	MEM OUT	**–**	2.27
dp12s12.shuffled.cnf**(*)**	SAT	15719	49	**320.80**	19.90
ezfact48-5.shuffled.cnf	UNSAT	65	47	**1.38**	0.65
glassy-sat-sel-N210-n.shuffled.cnf	SAT	20	27	**0.74**	0.10
grid-10-20.shuffled.cnf	UNSAT	509	508	**1.00**	0.31
hanoi5.shuffled.cnf	SAT	13027	1335	**9.76**	1.60
hanoi6-fast.shuffled.cnf	SAT	275	263	**1.05**	1.34
homer11.shuffled.cnf	UNSAT	652	489	**1.33**	1.42
homer12.shuffled.cnf	UNSAT	1944	548	**3.55**	3.24
ip38.shuffled.cnf	UNSAT	4680	1439	**3.25**	3.75
lisa20-1-a.shuffled.cnf	SAT	20	36	**0.55**	0.75
lisa21-3-a.shuffled.cnf	SAT	508	80	**6.32**	5.32
qg2-8.shuffled.cnf	SAT	110	19	**5.69**	0.80
rand-net50-60-5.shuffled.cnf	UNSAT	5605	720	**7.79**	9.42
vda-gr-rcs-w8.shuffled.cnf	SAT	364	399	**0.91**	2.10
w08-14.shuffled.cnf	SAT	30145	2287	**13.18**	7.58
w10-75.shuffled.cnf	SAT	162	143	**1.13**	2.01
pyhala-braun-sat-30-4-02.shuffled.cnf	SAT	7	38	**0.19**	0.21

Table 3. Experimental results for the GridSAT set of problems

#	OS	CPU	RAM
15	Linux 2.4.21	Intel Xeon 2.6 GHz	2GB
18	SunOS 8	UltraSPARC-IIe 500MHz	512MB
3	SunOS 8	UltraSPARC-III 750MHz	512MB
1	Windows 2000	AMD Athlon 900MHz	512MB

Table 1. Client Platforms

The time interval for reporting a client certificates was set to 10 seconds, which seemed to be a good compromise between network load and timely scheduling. The pool of subproblems had a minimum size of four, which is sufficient to only run out at the beginning and the end of the solving process. We enabled checkpointing of lemmas of a maximum length of 10. This client-side checkpointing was done every 15 minutes. Server-side checkpointing is an integral part of the system and cannot be disabled.

To level out different run times due to timing properties of our algorithm, we ran each benchmark suite 3 times and report an average value.

6.1 Benchmarks

For evaluating the ZetaSAT system we chose benchmarks whose sequential run time is within practical bounds, but large enough to be relevant. We also decided to use whole suites of benchmarks to give meaningful results for one problem class. We chose a suite of Bounded Model Checking [3] benchmarks found in the IBM Formal Verification Benchmark Library [22]. The '19_rule'-suite was taken, because it was the one that we could solve sequentially almost exhaustively (we only gave up on k100), but also contained challenging instances. The other set of benchmarks we picked is the Pyhala-suite of encodings of circuits that factor primes (UNSAT) or multiply two primes (SAT). They are taken from the collections of the 2002 and 2003 SAT competitions [30]. We do only report on long running instances (with a sequential run time above 3 minutes), because these instances are the target of ZetaSAT. Finally we chose to run the benchmarks reported on in the paper on GridSAT [10], because parallel run times and speed ups were given. We expected to find similarities, since GridSAT also employs the zChaff SAT solver.

6.2 Results

The results (Tables 2 and 3) clearly show the usefulness of our approach. The IBM benchmark suite as well as the Pyhala-suite show considerable speed-ups. The large hardware verification benchmarks with up to a million clauses were successfully solved. The biggest instance we considered (k95) exhibited a speedup greater than 7.

Especially the large UNSAT instances of the Pyhala-suite show a good speedup (up to 40). This is a valuable result as it shows that our system is not only a 'model finder', like e.g. randomized incomplete SAT algorithms, but can also speed up the proof of unsatisfiability.

On the GridSAT set of benchmarks there are speed-ups for the long-running instances as well. However, as it is not mentioned in [10], how their solver exactly was configured and what hardware was used in detail, one cannot compare the numbers with the GridSAT results. In fact already our sequential run times are very different. In Table 3 we cite GridSAT speed-up numbers only to give another example on the range of speedups achievable.

On two instances we could not get a result with ZetaSAT, because the system ran out of memory. This is clearly a problem and we will address memory shortage in further research.

The super-linear speed-ups obtained for the marked satisfiable instances can be attributed to search anomalies common in this kind of parallelization [20]. The reason for this is that through splitting, search space will be covered in a different order than by the sequential solver.

7 Conclusion and Future Work

In this paper we reported on our Desktop Grid enabled Boolean satisfiability checker ZetaSAT. The main contribution of our paper is that we have demonstrated that highly irregular constraint satisfaction problems (which are typically based on backtracking search and employ sophisticated heuristics for pruning the search space) can be beneficially deployed in Desktop Grid environments.

An important objective of our future work will be to design a more generic platform, facilitating the deployment of all kinds constraint satisfaction problems on Desktop Grids.

As stated in our introduction, SAT is a universal tool for solving problems of a wide range of practical disciplines. In this context, another future goal is to provide a more abstract framework for realizing application specific SAT based Desktop Grids. In this context, another focus of our future work will be to integrate a more advanced dynamic learning scheme into ZetaSAT. In [6] we describe a technique for distributed dynamic learning which is based on mobile agents. In order to adopt this scheme for Desktop Grids, we have to realize a Desktop Grid platform which has full peer-to-peer functionality, enabling mobile agents to travel around the compute clients on order to gather and exchange pertinent lemmas.

References

[1] L. Baptista and J. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proc. of the 6th International Conference on Principles and Practice of Constraint Programming (CP)*, 2000.

[2] R. J. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. of the 14th National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS. Springer-Verlag, 1999.

[4] W. Blochinger, W. Küchlin, C. Ludwig, and A. Weber. An object-oriented platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999.

[5] W. Blochinger, C. Sinz, and W. Küchlin. Parallel consistency checking of automotive product data. In G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, editors, *Proc. of the Intl. Conf. ParCo 2001: Parallel Computing – Advances and Current Issues*, pages 50–57, Naples, Italy, 2002. Imperial College Press.

[6] W. Blochinger, C. Sinz, and W. Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.

[7] W. Blochinger, C. Sinz, and W. Küchlin. A universal parallel sat checking kernel. In H. R. Arabnia and Y. Mun, editors, *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications PDPTA 03*, volume 4, pages 1720–1725, Las Vegas, NV, U.S.A., June 2003. CSREA Press.

[8] M. Boehm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):381–400, 1996.

[9] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel Distr. Comp.*, 63:597–610, 2003.

[10] W. Chrabakh and R. Wolski. GridSAT: A Chaff-based distributed SAT solver for the grid. In *Proc. of Supercomputing 03*, Phoenix, Arizona, USA, 2003.

[11] S. A. Cook. The complexity of theorem proving procedures. In *3rd Symp. on Theory of Computing*, pages 151–158. ACM press, 1971.

[12] J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1092–1097, Seattle, Washington, 1994. AAAI Press/MIT Press.

[13] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[14] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[15] The Folding@home project. *http://www.stanford.edu/group/pandegroup/folding*.

[16] G. C. Fox and W. Furmanski. High-performance commodity computing. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a new computing Infrastructure*, chapter 10, pages 237–256. Morgan Kaufmann Publishers, 1999.

[17] The Genome@home project. *http://www.stanford.edu/group/pandegroup/genome*.

[18] C. Germain, G. Fedak, V. Neri, and F. Cappello. Global computing systems. In *Large-Scale Scientific Computing : Third International Conference, LSSC 2001, Sozopol, Bulgaria. Revised Papers*, number 2179 in LNCS, pages 218–227, Berlin, Heidelberg, 2001. Springer-Verlag.

[19] The great internet mersenne prime search (GIMPS) project. *http://www.mersenne.org*.

[20] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):28–35, 1999.

[21] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[22] IBM Formal Verification Benchmark Library. *http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/benchmarks.html*.

[23] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proc. of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[24] D. Kondo, M. Taufer, J. Karanicolas, C. L. Brooks, H. Casanova, and A. A. Chien. Characterizing and evaluating desktop grids: An empirical study. Technical Report 2003-0769, UCSD-CSE, 2003.

[25] J. P. Marques-Silva and K. A. Sakallah. Grasp - a new search algorithm for satisfiability. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, 1996.

[26] F. Massacci and L. Marraro. Logical cryptanalysis as a SAT problem. *Journal of Autom. Reas.*, 24(1-2):165–203, 2000.

[27] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the 38th Design Automation Conference*, 2001.

[28] SETI@home project. *http://setiathome.ssl.berkeley.edu*.

[29] M. R. Shirts and V. S. Pande. Screen savers of the world, unite! *Science*, 290:1903–1904, 2000.

[30] L. Simon and D. L. Berre. SAT Competition 2003, 2003. http://www.satlive.org/SATCompetition/2003/comp03report.

[31] P. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. Combinational test pattern generation using satisfiability. *IEEE Transactions on Computer-Aided Design*, 15(9):1167– 1176, 1996.

[32] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major seti project based on project serendip data and 100,000 personal computers. In *Proc. of the 5th International Conference on Bioastronomy*, 1997.

[33] M. N. Velev and R. E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Proc. of the 38th Conference on Design Automation Conference*, 2001.

[34] ZetaGrid. *http://www.zetagrid.net*.

[35] H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journ. Symb. Comp.*, 21:543–560, 1996.

[36] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. of ICCAD*, San Jose, CA, 2001.

[37] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. of 8th International Conference on Computer Aided Deduction*, 2002.