Solving "Hard" Satisfiability Problems Using GridSAT *

Wahid Chrabakh (chrabakh@cs.ucsb.edu)
Rich Wolski (rich@cs.ucsb.edu)
University of California Santa Barbara,Department of Computer Science, Santa Barbara, CA, USA

September 1st, 2005

Abstract. We present the latest instantiation of GridSAT (Chrabakh and Wolski, 2003), a distributed and complete satisfiability solver that is explicitly designed to aggregate grid resources for application performance. GridSAT was previously shown to outperform the state-of-the-art sequential solvers. In this work, we explore the unprecedented solving power GridSAT enables through algorithmic and implementation innovations. We describe the implementation techniques that allow GridSAT to leverage a variety of high-end batch-scheduled resources, clusters, interactive workstations, and personal computing resources through autonomous scheduling, checkpoint scheduling, and work migration. These innovations have allowed GridSAT to solve a set of "hard" and previously unsolved industrial and community satisfiability problems. In addition to this new solution power, GridSAT also outperforms the otherwise highest performance general solvers on the annual SAT competition (SAT Competition, 2005) performance benchmarks.

Keywords: Parallel, Distributed, Scheduling, Satisfiability, Computational Grid

1. Introduction

Grid computing (Foster and Kesselman ed., 1998; Berman et al ed., 2002) is an emergent field in computer science that focuses, in part, on the aggregation of geographically distributed and federated computational resources. These resource aggregations can be harnessed by grid applications to solve problems in science and engineering (Wilfred et al., 2004; Gabrielle et al., 2001) which require large computing power. Solving such challenging problems and enabling new scientific results is an integral part of the grid computing vision.

One such challenging problem is propositional satisfiability. This problem involves finding a set of binary assignments to variables that satisfies a set of constraints (i.e. makes a binary expression evaluate to "true"). The problem of solving satisfiability instances is important from both theoretical and practical perspectives and is, in general, NP-complete. In practice, many engineering disciplines require

* This work was supported by grants from the National Science Foundation, numbered CAREER-0093166, EIA-9975020, ACI-0103759, and CCR-0331654 and by the San Diego Supercomputer Center and the TeraGrid project.

the solution to domain specific instances of satisfiability. Such disciplines include scheduling (Ramn Bjar and Felip Many, 2000), model checking (Li et al., 2003), security (Alessandro Armando and Luca Compagna, 2003), Artificial Intelligence (Henry Kautz and Bart Selman, 1992), software verification ("D. Jackson and M. Vazir, 2000), and the the area of Electronic Design Automation (EDA) which includes circuit design (Silva, 1995), Field-Programmable Gate Arrays (FPGA) detailed routing (Nam et al., 2001), combinational equivalence checking (Kunz and Stoffel, 1997) and automatic test and pattern generation (Larrabee, 1992).

As a result, an extensive body of research has focused on the development of highly-efficient satisfiability solvers (Moskewicz et al., 2001; Goldberg and Novikov, 2002; Hirsch and Kojevnikov, 2001; Biere, 2001). The results of this research have lead their wide-spread use in industrial and research settings where the solution to general satisfiability problems is essential (Silva, 1995; Nam et al., 2001; Li et al., 2003). These solvers use different techniques to navigate the entire search space of possible truth assignments for the variables of a given expression. The best (fastest and most comprehensive) of these solvers use *learning* optimizations that permit the search space to be "pruned" during execution. Learning (Silva and Sakallah, 1996) introduces new deduced propositions which improve the solver's efficiency by obviating subtrees in the space of possible variable assignments.

Because learning requires a large, centralized database of intermediate propositions to be searched and updated frequently, the most successful solvers resulting from prior work are sequential. These sequential solvers are characterized by heavy use of compute power (CPU) as well as the memory of the host machine as the database must be kept memory-resident (or the speed becomes unacceptably low).

Research in parallel solvers (Chrabakh and Wolski, 2003; Jurkowiak et al., 2001; Sinz et al., 2001; Forman et al., 2002), shows that using a large pool of computational resources can lead to better performance for some problems. The aggregate CPU power and memory of the hosts allows the solver to navigate the search space faster. However, because the clause database used in learning is accessed so frequently, with the exception of (Chrabakh and Wolski, 2003), these initial parallel efforts have not been able to take advantage of learning optimizations in a distributed environment. Thus the fastest solutions to the largest number of problems (i.e. the most generally successful approach) have been achieved by sequential solvers that employ learning (SAT Competition, 2002,2003) prior to our work.

By carefully leveraging the resources in grid settings, our goal is to build a parallel and distributed satisfiability (SAT) solver that correctly determines the solution to previously infeasible industrial problem instances, the answers for which cannot be determined in any other way. Secondarily, we would like to be able to improve upon the fastest time to solution for problems that have previously been solved by either parallel or sequential solvers.

Our previous work with GridSAT (Chrabakh and Wolski, 2003; Chrabakh and Wolski, 2003) demonstrates the latter. By dynamically acquiring and releasing resources under the control of an automatic scheduler, GridSAT improves the time to solution for various feasible SAT instances. Indeed, GridSAT outperforms the previously best-performing solver on all problems that this leading solver can complete (SAT Competition, 2002,2003). We have also been able to use GridSAT to solve several previously unsolved problems using non-dedicated, wide-area grid resources. It is these new domain-science results, and the techniques we have employed to achieve them, that are the subject of this paper.

In particular, by combining different batch-controlled super-computers with interactive workstations and user desktop machines, we have applied GridSAT to *hard* SAT problems – ones that are not only unsolved but for which previous attempts at solution using other general techniques have failed. This pattern of combining different types of resources is new and different from that used by existing *parallel* SAT implementations (Jurkowiak et al., 2001; Sinz et al., 2001). Moreover, we know of no *distributed* (i.e. network and/or grid enabled) SAT implementations, efficient or otherwise, at the time of this writing.

Distinct resource types are utilized in different ways during application execution. In general, the resources in a computational grid may be of two different types: time-shared or batch controlled. In the case of time-shared resources the application will compete with other user applications running simultaneously on the host machine. However, since these resources are always available the application can continue to make progress. Other resources which are controlled by a batch scheduler, will participate intermittently in the application through some of their nodes. But these systems will provide significant compute power depending on the size of the application's request.

In order to enable a grid implementation of a SAT solver to use many resources simultaneously, we need to address two types of challenges. First the solver's algorithm needs to be modified so that it can run in parallel while ensuring that the parallel components cooperate to improve over-all efficiency. Using a parallel algorithm makes it possible to reason about additional optimizations which were not possible in the sequential case. These optimizations are related to sharing intermediate results between parallel components during execution. The second challenge is developing a framework capable of running the parallel solver in a very volatile computational environment while maintaining overall solver efficiency.

Solving the above two problems was at the core of our methodology in designing the application components and their interactions. Implementing this methodology can be achieved by selecting suitable technologies. Examples of these technologies include those from parallel computing, which predate grid computing, such as MPI (MPI, 1994). The more relevant technologies are those which were the outcome of grid-specific research projects such as Globus (Foster and Kesselman, 1997), Web Services (W3C, 2003) and related standards. We discuss in this paper the requirements imposed by the application's dynamic behavior and constraints on the technology so that a successful implementation is realized. We also describe the current design and implementation of the application.

Using the above methodology we have developed GridSAT, a distributed satisfiability solver capable of running on a computational grid. GridSAT implements a parallel algorithm for solving satisfiability problems based on Chaff (Moskewicz et al., 2001). GridSAT distributes and shares the internal proposition database among processors in a way that takes advantage of dynamic resource performance predictions to achieve new levels of solver efficiency.

In this paper, we detail the current, most capable version of GridSAT. Our most recent improvements in the clause sharing and resource scheduling algorithms have made it possible to solve previously unsolved satisfiability problems from the field of FPGA routing as well as artificially generated benchmarks specifically designed to foil automatic SAT solvers.

The paper is organized as follows. In section 2 we present GridSAT's parallel version of the algorithm and the improvements added over previous implementations. Section 3 presents the GridSAT architecture and scheduler. The implementation methodology is detailed in section 4. We present experimental setup and results in section 5. Finally, we conclude in section 6.

2. GridSAT: SAT Solver for the Grid

A satisfiability problem is expressed as a boolean formula over a set of variables. Most solvers operate on formulas expressed in Conjunctive Normal Form (CNF) in which an expression conjoins (logically "ANDs") a set of *clauses*, each of which may contain disjoined ("ORed") literals. A literal is either an instance of a variable (V) or its complement ($\sim V$) and variables are boolean. A SAT problem instance is termed *satisfiable* if there exists a set of variable assignments that makes the formula evaluate to *true* where "true" corresponds to a boolean 1 algebraically. If such an assignment does not exist the the problem is declared *unsatisfiable*.

GridSAT is based on Chaff (Moskewicz et al., 2001), a sequential SAT solver algorithm. Chaff, in turn, builds upon the DPLL (Davis-Putnam-Loveland-Logemann) (Davis et al., 1962) algorithm which solves a SAT instance by making a set of speculative variable assignments (termed "decisions") stored in a *decision stack*. When these decisions are propagated through the clauses they could lead to a cascade of *implications*. Implications are assignments of boolean values to different variables as deductive consequences of previous speculative decisions. These speculative decisions and the resulting implications may lead to logical conflicts – deduced contradictions in which a variable must take on both boolean values because of different clauses in the original problem. In Chaff, as well as other solvers, the performance of the algorithm is enhanced by using techniques for adding new deduced clauses after a conflict occurs. This technique is called *Learning* (Schulz and Auth, 1989; Larrabee, 1990; Silva and Sakallah, 1996). Using learning, the algorithm may generate a vast number of additional clauses during execution. These clauses consume memory, possibly overwhelming the capacity of the host, and also may slow the algorithm as they can add to the search complexity of the clause database.

GridSAT's distributed solver addresses three significant challenges to improving solver performance. First, GridSAT parallelizes the search algorithm that is navigating the space of possible truth assignments. Second, certain learned clauses from the various solvers are selected to be distributed and shared across resources. Finally, the GridSAT application components are dynamically scheduled at runtime to take advantage of those available resources which can enhance the solver's performance.

To apply a parallel search technique to SAT, we split the original problem into subproblems (having decision stacks with different truth assignments), each of which is independently investigated for satisfiability. Subproblems, themselves, may be split in the same way, forming a recursive tree, each node of which is assigned to a logically distinct processor. Clause sharing is facilitated by identifying and sharing only important clauses.

2.1. Sharing Learned Clauses

Each client can share newly learned clauses with other clients to help prune their search spaces further. Fortuitously, smaller clauses have the greater effect on pruning the search space (Chrabakh and Wolski, 2003), but the point in the algorithm in which new clauses are considered can affect the efficiency of clause sharing. We consider three different methods for merging shared clauses after they are received by a remote processor: the *lazy*, *immediate* and *periodic* methods.

The lazy method is the simplest to implement. The client stores received clauses until the local solver backtracks to the highest level in the decision stack. The advantage of this approach is that clauses are introduced without modifying the current decision stack "in-flight." However, if received clauses are considered earlier, then they will help direct the search and make the solver more efficient.

The opposite approach – the immediate method – allows clauses to be merged immediately after they are received. New clauses could cause implications or conflicts at higher levels in the decision stack. Therefore, the solver may need to backtrack as a result of clause introduction and modify the decision stack accordingly. Such updates to the decision stack are complex, both from a logical consistency point of view and in an engineering context. Moreover, if the introduction of new clauses does not trigger backtracking, the overhead associated with interrupting the solver and threading the new clauses into the database may actually retard performance.

As a compromise, the periodic method is designed to periodically merge clauses. At present, the periodicity is determined by a user specified parameter. This periodic method allows the solver to merge received clauses more frequently than the lazy method while merging clauses in batches and interrupting the local solver less frequently than the immediate method. In this paper we set the periodicity to 60 seconds, but we are exploring ways to automatically schedule the introduction of new clauses as a future enhancement to GridSAT.

3. GridSAT Architecture and Resource Scheduling

GridSAT is implemented as a special form of the coordinator/client model where individual clients communicate directly and share clauses (i.e. communication is between peers rather than routed through the master). The GridSAT application uses two views of the computational resources as shown in figure 1. The first view employs jobs to classify processes which belong to the same resource. The second view is flat

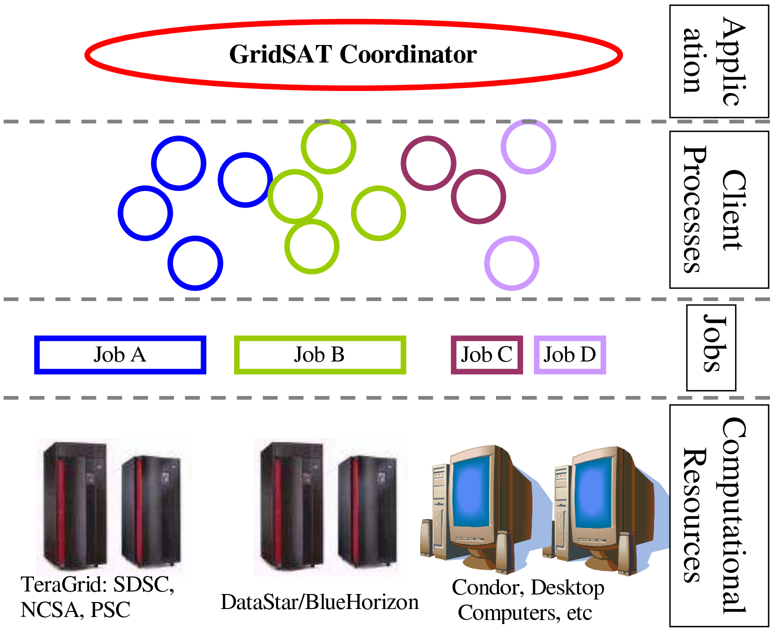

Figure 1. GridSAT resource views

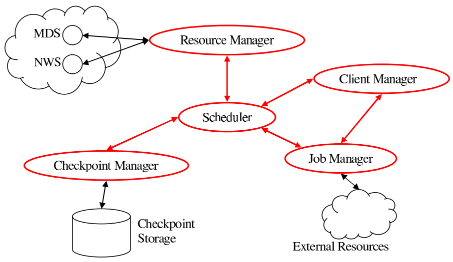

Figure 2. GridSAT components and their internal and external interactions. The external components and systems which GridSAT uses, such as the Globus MDS and the NWS, are shown in clouds.

where all processes are part of a single pool. Both of these views are useful for managing resources under GridSAT

The coordinator (or master), shown in figure 2, reflects the resource views shown in figure 1. It consists of the resource manager, the job manager, the client manager, the scheduler and the checkpoint server. We now describe the role of these components.

The resource manager is tasked with loading resource information from one or more grid information systems such as Globus MDS (Czajkowski et al., 2001) and the NWS (Wolski, 2003; Wolski et al., 1999). The scheduler, however, is responsible for coordinating the interactions between all the components. In addition, it handles interactions with external resources and monitors them to detect failures. For example, the scheduler queries the resource manager for resource types. If the resource is time-shared, then only one GridSAT process is launched. For batch systems, the scheduler instead submits one job request. Additional jobs could be manually submitted and GridSAT will use their resources when they become available. We term this form of scheduling *active queuing*; jobs waiting in queue logically execute on the interactive resources until the batch-controlled resources become available. At that time, the scheduler migrates work into the newly available resources. Thus, the application makes progress using the slower, shared resources while it waits in queue. The job manager is responsible for monitoring the status of all submitted jobs which may be active or queued. The client manager maintains a list of all GridSAT individual processes and monitors their progress.

The GridSAT scheduler is the focal point and is responsible for coordinating the rest of the components and launching new processes, also termed clients. The scheduler uses a progressive scheme for starting additional clients on remote resources and adding them to the active resources' pool. Resources which are no longer performing a task on behalf of GridSAT are released immediately when possible. The reason for this approach is the variability and unpredictability of resource usage for a particular SAT problem. Some problems are solved easily using a single host after a short time period. Other problems, however, might be harder and require a large number of hosts and a longer time period. By starting with a small resource pool and expanding the set of used resources, GridSAT achieves three goals. First, a small number of resources will be used to solve the easy problems which results in a smaller communication overhead and therefore shorter time to solve the problem. Second, GridSAT can adapt its resource usage to how difficult it perceives the problem to be. For examples, if the problem is perceived difficult at a particular stage, then the the size of the resource pool GridSAT uses will grow. At another stage, the same problem might be perceived to be easy and a smaller resource set will be used with excess resources released. Lastly, by remaining as small as possible at any given point in the execution, GridSAT promotes allocation stability and sharing. The scheduler does not waste resources needlessly thus the maximum number of GridSAT instances can co-exist since each is attempting to use as few resources as possible for its own problem instance.

The GridSAT scheduler uses the first available client immediately to start solving the problem. The decision for splitting a problem is made locally by the client and not by a centralized scheduler. The client makes this decision by using two measurements. First, each client records the time it took to receive the problem data. Second, individual clients monitor their memory usage throughout their execution period. Using both of these parameters, a client notifies the master that it wants to split its assigned subproblem with another client when its memory usage exceeds a specified limit (currently 80% of forecast available memory which it determines from the NWS) or after running for a specific period of time. This time period is determined as twice the duration of the the communication period the client used to obtain the problem data. Using this method, the scheduler allows for computation time to offset the communication overhead by using the previous communication period as a prediction of future overhead. The clients, therefore, do not spend most of their time splitting instead of doing useful computation. The splitting process is performed by the cooperation of the master, the splitting client and an *idle* client. The *idle* client is a process which is not currently assigned a subproblem to investigate.

The GridSAT solver terminates when all subproblems are solved or one of the clients finds a satisfying assignment. In the latter case the client which finds the satisfying assignment sends its solution state (in the form of an "assignment stack") to the master. Finally, the master saves the final solution, terminates all running clients and cancels any pending resource requests. Most solvers in the literature are evaluated based on the time the first satisfiable instance is found. However there are cases where knowing all satisfiable instances is desired ("D. Jackson and M. Vazir, 2000). GridSAT can also enumerate all the instances where a problem is satisfiable although clearly at a greater resource cost that is has the potential to be significantly greater than for a single solution.

3.1. Active Queuing: Efficient Use of Batch Jobs

Batch controlled systems are usually very powerful supercomputers and clusters which can provide large computational power. Because the GridSAT application has dynamic resource requirements, the scheduling policy adopted by GridSAT aims at accomplishing two goals. First, these resources must be used efficiently so that allocated processors spend very little time idle when batch jobs start executing. The second goal is to use batch jobs only when it is justifiable for the application to use such large computational power. Since GridSAT has variable resource requirements throughout runtime the policy is designed so that large batch jobs are only used when it advantageous to solving the current problem.

Initially, the GridSAT scheduler submits batch job requests that are large with a high number of nodes and long duration. This leads to a long waiting period in the scheduler's batch queue. Thus, if a job is not solved after this long waiting period, then it most probably is a hard problem. Therefore batch jobs are only used when the problem is hard. When a batch job starts execution, GridSAT migrates work (as a checkpoint file) to achieve more efficient use of batch nodes. Remote GridSAT nodes, which are numerous, will migrate their work immediately to occupy batch nodes. After migration takes place and since networks are fast within super-computing nodes, splitting happens at higher rates especially after the above mentioned reductions in communication overhead. Moreover the GridSAT scheduler senses the additional bandwidth between clients executing on a supercomputer or cluster. It then increases the size and number of clauses shared by subproblems inside the tightly coupled resource as a further improvement. Thus, through a combination of *introspection* and the NWS, GridSAT automatically senses the performance topology available to it and reorganizes itself to take the best advantage of the resources at hand. Note that the number of active nodes (i.e. those with subproblems) will increase exponentially. This happens because the number of new subproblems is increased in proportion to the number of existing active solvers.

3.2. Components Design

There are two types of components in GridSAT: coordinator and client. Both components have similar basic designs even though they perform different functions. The basic features they share is that they are both threaded and handle asynchronous communication. These features allow each component to perform different tasks simultaneously. Also when one task takes a long time to process other tasks are not starved. For example, if a remote resource experiences failure while the coordinator is sending to, or receiving a message from, this resource, the communication timeout would be very long. If the application was single threaded or did not handle asynchronous communication, the coordinator would not be able to communicate with any other resource until the failed resource times out or recovers. Furthermore, EveryWare (Wolski et al., 1999) –the communication system currently used by GridSAT– provides additional reliability and performance through adaptive time-out discovery.

In GridSAT, we have identified three types of messages which are managed differently by the application components. These message types are categorized based on their scope and reliability requirements. The three message types are:

- Control Messages: These messages are sent from the coordinator to the clients or vice versa. There are many such messages but overall these messages are sent intermittently. Thus they do not represent a high communication overhead. These messages are delivered directly between the clients and the coordinator.

- Broadcast Messages: Broadcast messages are sent from the coordinator to the clients or from a specific client to all the other clients. In the first case, the coordinator broadcasts messages only when it asks all clients to terminate or when a specific solver parameter is to be globally modified. In the second case, however, a client who obtains an intermediate clause which satisfies the sharing criteria, would want to share this result with all other clients. In order to minimize communication overhead, a tree structure is established where the root is at the sender and the leafs are the recipients. GridSAT uses a single tree where the initial root is the coordinator. This tree can trivially be used to broadcast messages originating at the coordinator from top to bottom. When a client broadcasts a messages it assumes that it is the root of the same tree and messages are broadcast along the same edges but in opposite direction is some cases. The tree is constructed dynamically where each job represents a new branch in the tree. The first client in that job is responsible for forwarding messages emanating to or from the job. This nodes is called the hub-node. The broadcast tree can also self-heal after node failures. If the hub-node fails another client from the job is chosen to be the hub-node. Once the new hub-node is selected, the coordinator directs the rest of the clients in the job to use the new node to forward their broadcast messages.

- Peer-to-peer Messages: Peer-to-peer messages are used by GridSAT clients in order to communicate directly without using the coordinator as an intermediary. This type of messages is used to reduce the communication and processing load on the coordinator node. More importantly it is used to reduce communication overhead for very large messages. Some the messages exchanged by GridSAT clients are very large and can be more that 0.5 GB in size. Therefore, direct peer-to-peer communication makes such message transfer faster (especially when clients are running inside a machine with a fast network and the coordinator is located outside that machine). However, this form of communication needs to be setup by the coordinator so that the sender is aware of the recipient of a given message. In the setup process the coordinator uses the smaller control messages to disseminate the required information to the interested clients.

The implementation of the messaging system for all three types of communication is similar. A message can be delivered through any of the messaging types by simply changing the message type when the message is initialized.

In addition, the message types described above are general and can be used to implement other grid-applications which are similar to GridSAT. Such applications include branch-and-bound like applications as well as master-client applications.

4. Grid Implementation

4.1. Application Characteristics

The GridSAT application is different from most high-performance computing applications in terms of programming model and resource usage. In general, the programming model for these applications is characterized as a set of alternating steps involving computation and communication. In addition, the computation and communication intervals do not overlap. These communication steps are also used as synchronization barriers which enable the various components of the application to exchange information. From the resource usage perspective, these applications use a predetermined set of compute resources throughout their execution.

Our application differs in much of the above aspects. The GridSAT application has variable resource requirements depending on the problem instance. The number of resources and duration of use of those resources cannot be predicted in general for satisfiability instances. In fact, the set of active resources which are assigned parts of the search space during runtime is dynamic. On the one hand, resources are added each time the problem is split. On the other hand, resources are released immediately after a subproblem is solved. At any given instance, there can be many simultaneous acquiring of new resources, through problem splitting, and release of other unneeded resources. Moreover, the application components share intermediate results as soon as they are produced. These results are asynchronously used by all the receiving clients.

Therefore, all the GridSAT segments are event driven and events are produced and consumed asynchronously. The solver components, for instance, can simultaneously perform communication and computation. All application modules are designed and implemented to allow for efficient management and responsiveness to these events.

Dynamic resource usage has been shown to help solve efficiently a large set of satisfiability problems (Chrabakh and Wolski, 2003).

Solving "hard" satisfiability problems represents further challenges. For "hard" problems, a small number of resources would be exhausted in a relatively short time. The CPU and memory resources would be saturated and additional resources are required in order to make progress in solving the problem under investigation. Therefore, we wanted to use all computational resources at our disposal, in order to render the solution of the hardest problems more plausible. The set of available resources varied from desktop machines, to small-size clusters, to supercomputers. This collection of resources was heterogeneous in terms of hardware, Operating Systems and resource management software. This heterogeneity represents a further challenge to the deployment of the application.

These application characteristics described above represent a true Computational Grid application. As "power" is added to the grid GridSAT can access, it makes efficient use of that power without undue waste. At the same time, fluctuations in available power are tolerated automatically so that the overall application remains maximally efficient while it is executing. Thus, GridSAT is one of the first programs to realize the vision of grid computing originally articulated in (Foster and Kesselman ed., 1998) and to demonstrate this capability by generating new domain science. Moreover, these characteristics are not unique to GridSAT. Other branch-and-bound or coordinator-worker applications can benefit from a similar use of computational resources.

A major challenge before implementing the various application components was to develop an implementation strategy. The final implementation aims at using all the available grid resources efficiently while dynamically adjusting to the application behavior and resource needs.

4.2. Implementation Strategy

Given these resource usage patterns, which are typical for a true Grid application, we had to choose an implementation strategy which would satisfy these requirements. There are several technology choices to select for the implementation of the application. Such options include, among others, MPI (MPI, 1994), Globus (Foster and Kesselman, 1997), vanilla Web Services (W3C, 2003) and later improvements such as WSRF (OASIS, 2003).

According to our experience with GridSAT we have learned that a successful implementation technology should allow for three pivotal capabilities: dynamic resource pool management, error detection/reporting, and universal deployment.

The first capability is to allow the use of a dynamic resource pool. This feature, for example, was not available in MPI-I which did not allow for dynamic Communicators. MPI-2 has introduced extensions to allow for dynamic creation and destruction of communicators. Globus and Web services also allow for a dynamic set of resources.

The second capability is error detection and reporting. Since GridSAT runs for extended periods of time using a set of geographically distributed resources, then network and resource failures are more frequent. Therefore in order to implement this application we need a technology which allows for the detection of these errors. From the perspective of the application, the distinction between resource and network failures is not important. It suffices for the application to obtain a feedback if a certain operation is not successful after a certain time period.

Error detection and recovery is very important because in our experience all resources experience a failure at some point. Even those resources which are professionally maintained can become unresponsive from the application's perspective. Those resources that do not experience hardware and software failures usually have scheduled routine preventive maintenance periods or a combination of software and hardware upgrades. From the point of view of the application these are "scheduled" or "anticipated" failures. Without rigorous error handling the application would not be able to run for extended periods as shown later in the results section.

Different technologies provide some form of error handling. MPI-I allows for error handling in a limited scope which is expanded further in MPI-2. Globus GRAM allows for error handling and call-back functions for job management. In Web Services, WS-Notification (Graham et al., 2004), WS-BaseFaults (Tuecke et al., 2005) and related standards could be used to provide this functionality. The desirable error handling for our application is to provide a time period for some actions after which some form of error handling should be performed. Sometimes if an action fails, then all that is needed is to retry it. In other cases, it is assumed that the resource (or the connecting network) has failed. This form of error handling is not available for the grid technologies mentioned above and can be implemented at the application level.

The last desirable capability for a suitable grid technology is universal deployment. This is not entirely a characteristic of the technology but of the computational environment as well. A widely deployed technology is advantageous because it reduces the development overhead since one version can be deployed on all available resources. In our experience there was no grid technology that was universally adopted and deployed which would enable us to combine all computational resources at our disposal. Thus a multi-infrastructure approach such as EveryWare (Wolski et al., 2001; Wolski et al., 1999) was necessary.

Furthermore, in order to deploy our application over a large set of resources,we had to interface with many types of resource managers. For example, resources could be managed by one of many Batch schedulers, Condor (Tannenbaum and Litzkow, 1995) or simply shared. Our goal was to use all these resources simultaneously regardless of what systems they originate from. This is accomplished by determining a general job description which can be instantiated differently using specific launchers for each resource manager. For instance, shared resources can be accessed directly using SSH. Batch systems, however, are accessed by submitting a batch script with syntax tailored to the scheduler used. Whenever, Globus is deployed we use it to launch and monitor job submissions.

4.3. GridSAT Implementations

We believe that many of these technologies could be used to develop GridSAT. In fact, we have developed a previous versions of GridSAT called GrADSAT (Chrabakh and Wolski, 2003) (note the "A" in the spelling) using *GrADSoft*. GrADSoft is a set of programming abstractions where the baseline grid infrastructure is provided by Globus and the NWS. GrADSoft is part of the **Gr**id **A**pplication **D**evelopment **S**oftware (GrADS) project (Berman et al., 2001; GrADS Web,) which is a comprehensive research effort studying grid programming tools and application development. To facilitate experimental application research and testing, the project maintains a nationally distributed grid of resources for use as a production testbed. Since the GrADS tools were universally deployed on this testbed we were able to deploy our application with little overhead on the entire testbed.

The current version of GriDSAT uses EveryWare (Wolski et al., 2001; Wolski et al., 1999) a very portable communication library. EveryWare has been designed explicitly to manage the heterogeneity and dynamism inherent in grid resource environments. EveryWare can be easily deployed as library on all the resources. In addition, all communication calls use a timeout argument, as desired, for error detection.

The resource management system interfaces with resources which use batch systems as well as desktop machines which are accessed through SSH. All resource related operations have been implemented to allow for a specific timeout. If the resource is not responsive after the timeout period expires, then the resource is considered unreachable.

In the future, we will explore other technologies as they become more widely used. Our goal would be to make GridSAT implementation independent where we can use an API for interfacing the application with the underlying communication infrastructure. As a result different grid technologies can be substituted without affecting the application.

5. Experimental Apparatus and Results

We present two sets of experiments. The first set relates to the algorithmic innovations introduced and compares the three clause merging methods described earlier. The second set demonstrates the ability of the GridSAT application to use a large set of heterogeneous and geographically distributed resources to solve previously unsolved problems.

5.1. Clause Merging Techniques

In this set of experiments we study the effectiveness of the three different learning methods: the lazy method, the immediate method and the periodic method. These methods use different algorithms to share intermediate clauses. The experiments are conducted using the set of 33 benchmark problems used by the different satisfiability competitions (SAT Competition, 2002,2003) and previous evaluations of GridSAT (Chrabakh and Wolski, 2003). The experiments were conducted on a set of dedicated nodes on a cluster available at UCSB. The cluster nodes are Pentium IV CPUs with 2.66 GHz frequency and 2 GB of memory. Each experiment uses ten nodes and one of the three methods. The experiments are grouped into three sets where the maximal size of a shared clause is varied between 5, 10 and 15.

5.1.1. *Results*

The experimental results are shown in table I. This table shows experimental results for using a maximal size of shared clauses of 5, 10 and 15 respectively. The table contains three sections, one for each size of shared clauses used. Each section shows to total time for each of the three methods and the relative speed-up compared to the lazy method.

In order to save space in this paper we omit the runtimes for the individual problems, But from inspecting each of the three experimental sets, we learned that no particular method outperforms the other two methods across all problem instances. To determine aggregate improvement, then, we use the total runtime of all the problems to compare the efficiency of the methods. Note that using the total runtime for all problems in a benchmark is the standard method for comparing solvers.

We notice that in each case both the immediate and periodic methods outperform the lazy method. The immediate method outperforms the lazy method by an average of about 7%. The periodic method was

Table I. GridSAT results comparing all three learning methods with maximal learn clause size equal to 5, 10 and 15. The total times are in seconds.

	Lazy Method	**Immediate Method**	**Periodic Method**
Maximum size of shared clause = 5			
Total	76776	68620	64675
% Speedup	*(base)*	10.6	15.8
Maximum size of shared clause = 10			
Total	71860	67292	63400
% Speedup	*(base)*	6.4	11.8
Maximum size of shared clause = 15			
Total	69527	67292	63400
% Speedup	*(base)*	3.2	8.8

the most efficient and showed a speedup of about 12% on average compared to the lazy method. We also notice that the speedup decreased as the size of maximal shared clause increased. These experiments show that using the periodic method gives the best overall performance by balancing the overhead of merging with the additional solver power that comes from clause sharing. In addition, since the periodicity is fixed at 60 seconds in this experiment, believe that further improvements are possible by explicitly scheduling the time and size of clause sharing.

5.2. Solving Hard Satisfiability Problems

Since GridSAT is a true grid application, (robust, portable, heterogeneous, pervasive, etc. (Foster and Kesselman ed., 1998)) we ran a set of experiments to show that GridSAT can run for extended periods of time robustly using a wide variety of resources and also solve previously unsolved hard satisfiability instances. In these experiments we simultaneously use computational resources that belong to collections of individual machines, small size research clusters and super-computing scale clusters. The computational resources we use are composed from four main sources:(1) 40 machines from the VGrADS (VGrADS Web, 2004) testbed located at UTK, UCSD and UCSB, (2) Blue Horizon at SDSC, (3) TeraGrid site at SDSC, (4) TeraGrid site at NCSA and (5) DataStar at SDSC.

The TeraGrid (TeraGrid, 2000) project is a multi-site national scale project which is aimed at building the worlds largest distributed infrastructure for open scientific research.

During our experiments, none of the resources we used were dedicated to our use. As such, other applications shared the computational resources with our application. It is, in fact, difficult to determine the degree of sharing that might have occurred across all of the available machines after the fact. In batch controlled system such as Blue Horizon, Data Star and the TeraGrid, the queue wait time incurred is highly variable because of jobs submitted by other users.

Thus, if it were possible to dedicate all of the VGrADS resources to GridSAT, we believe that the results would be better. As they are, they represent what is currently possible using non-dedicated Grids in a real-world compute setting.

These experiments also use a more diverse set of resources for longer periods of time (up to a month in duration) and multiple job requests. We chose a set of challenge problems from both (SAT Competition, 2002,2003) benchmarks. These benchmarks are used to judge and compare the performance of automatic SAT solvers at the annual SAT conference. All the problems in the benchmarks are shuffled to insure that submitted benchmarks are not biased in favor or against any solver. These benchmarks are used to rate all competing solvers. They include industrial and hand-made or randomly generated problem instances that can be roughly divided into two categories: *solvable* and *challenging*. The solvable category contains problem instances that some SAT solvers have solved correctly. They are used for comparing the speed of competing solvers. Alternatively, the challenging problem suite contains problem instances that have yet to be solved by an automatic method or which have only been solved by one or two automatic methods, but are nonetheless interesting to the SAT community. Some of these problems have known solutions that are known through analytical methods (i.e. the problem has a known solution by construction), but several of these problems are open questions in the field of satisfiability research.

In these experiments, we only chose problems from the challenging set. These problems were deemed hard by all participating solvers in both the 2002 and 2003 SAT competitions. We investigate seven previously unsolved problems where three instances are from the SAT 2003 benchmark category, and four are instances from the SAT 2002 benchmark category, all of which we have not been able to solve using previous versions of GridSAT.

This group of problems represent a variety of fields where problems are reduced to instances of satisfiability and solvers are used to deter-

Table II. GridSAT results using VGrADS testbed, Blue Horizon, Data Star and TeraGrid. All these problems were not previously solved by any other solver.

File name	SAT/UNSAT/*	Time	GridSAT Result
3bitadd-31(T)	UNSAT	8 days	-
k2fix-gr-rcs-w8(F)	*	83261 sec (23 hours)	UNSAT
k2fix-gr-rcs-w9(F)	*	14 days and 8 hours	UNSAT
cnt10(F)	SAT	13134 sec (4hours)	SAT
comb1(M)	*	11 days	-
f2clk50(M)	*	9 days	-
hanoi6(T)	SAT	23 days	-

(*): problem solution initially unknown
(T): Theoretical
(F): FPGA Routing
(M): Model Checking

mine the solutions. The problems contain a pair of problems in FPGA routing and model checking. These two disciplines benefit heavily from efficient SAT solvers. The remaining problems are of theoretical nature. In addition, we set the absolute minimum size of shared clauses to two and absolute maximum to 15. This range allows for sharing clauses which would help prune the search space without significant communication overhead.

Unlike previous experiments there was no timeout value set for the maximum execution time. Every problem was run using different job description for the batch systems. Jobs on the different batch queues were manually re-launched at random intervals. Job re-submission could have been automated but we wanted more control over rationing our limited compute budgets to specific experiments based on their perceived progress. Experiments where GridSAT was making progress were allotted bigger jobs with longer durations and more nodes. The progress of the solver was judged by inspecting how often the checkpoints were updated. We can also inspect the internal state of a particular solver using some of the tools we developed. The VGrADS nodes were used during the entire duration of each experiment unless the hosts experienced failures.

5.2.1. *Results*

The experimental results are summarized in Table II. The first column contains the problem file name. The second column indicates the field from which this problem instance in obtained. The third column

Table III. Batch jobs used to solve the k2fixgrrcsw9.cnf instance from SAT 2003 benchmark

Compute resource	Job count	Job dur.(hr)	Node count	procs /node
BlueHorizon	2	10	100	3
Blue Horizon	1	12	100	3
DataStar	2	10	8	11
TG@SDSC	1	10	40	2
TG@SDSC	1	12	40	2
TG@SDSC	3	10	4	2
TG@SDSC	4	5	4	2
TG@NCSA	3	10	4	2
TG@NCSA	4	5	4	2

in addition to 40 machines from VGrADS testbed for 14 days 7 hours and 44 minutes

contains the solution to the instance: satisfiable (SAT), unsatisfiable (UNSAT), or unknown. We have marked those problem instances which were previously open satisfiability problems with an asterisk (*). If a problem was originally unknown and was later solved by a solver, then we still keep it marked with an asterisk for completeness. The fourth column represents the total wall-clock time that the problem was tried. Finally, the fifth and last column represents the solution obtained by GridSAT which is represented by SAT, UNSAT or (-) if we terminated the experiment before GridSAT found an answer. Note that while we terminated these problem instances manually so that we could complete this paper, each can be continued from its last checkpoint (which we have archived).

Table II shows that GridSAT was able to solve three problems all of which were not previously solved. Two of the problems were found unsatisfiable and they are both from the field of FPGA routing. The first problem *k2fix-gr-rcs-w8.cnf* was solved using the VGrADS testbed only. Batch jobs which were submitted for this experiment were canceled when the problem was solved. On the other hand the second problem *k2fix-gr-rcs-w9.cnf* took much longer to solve, it took more than two weeks. Table III gives a more detailed description of the resource used during this experiment. For each job a number of GridSAT solver components were launched as indicated in the last column of table III. In table 5.2.1 a break down of the CPU-hours used on each resource are tabulated. Note that the VGrADS testbed machines were

Table IV. CPU-hours per resource used to solve the k2fix-gr-rcs-w9.cnf instance from SAT 2003 benchmark

Compute resource	node- -hours	CPUs/ node	CPU -hours
BlueHorizon	3200	8	25600
DataStar	160	11	1760
TG@SDSC	1080	2	2160
TG@NCSA	200	2	400
GrADS(*)	13750	1	13750

(*) machines were shared with other users

able to deliver a sizable amount of compute power because they were available in a shared mode for the duration of the experiment.

The last problem *cnt10* was also solved using the VGrADS testbed only under similar circumstances to *k2fix-gr-rcs-w8*. We previously tried solving this problem in (Chrabakh and Wolski, 2003) using the same testbed for four days in addition to Blue Horizon for 12 hours but were not successful. We believe the improvements made to the solver and especially the new clause sharing method have helped achieve this result.

In order to illustrate further GridSAT's success in using all the above variety of resources mentioned earlier we present a section of a run using instance *hanoi6*. This problem is a SAT representation of the *Hanoi Towers* problem using six disks. A six day snapshot from a 23 day run is shown in figure 3 using logarithmic scale. The figure shows several jobs from Blue Horizon, Data Star and TeraGrid sites participating in the execution. This figure shows that GridSAT was able to make use of the available resource when some of their nodes became available and then continued to run after the nodes were taken away to serve other users. GridSAT processes continue to run on the batch controlled resources until the scheduler decides to terminate them. This abrupt termination has no effect on the application which deals with these events as (scheduled) resource failures. GridSAT was able to manage up to 350 processes running on different resources as show in this figure.

The satisfiability solver performs mostly integer, branching and load-store operations. The number of floating point operations is very low (less than .1 FLOPS). We present in figure 4 an estimate of the total number of instructions per second during the same six day period. Since instrumenting GridSAT can cause significant slow down, we conducted some benchmarking on some machines at UTK to determine the average efficiency of the solver. Since the solver code is mostly sequential, we assume that at the maximum only one instruction per cycle can be finished by the processor. The determined efficiency is 70%. We estimated that other hardware and OS combinations will exhibit equal efficiencies. The number of operations provided by a resource is estimated to be the product of its peak performance and the estimated efficiency. The total number of instructions in figure 4 is the sum of operations of all active resources. We notice that the VGrADS testbed is able to deliver about 20 *Billion instructions per second* (IPS). In the middle of the graph, there is a batch job from Blue Horizon which failed suddenly while joining the GridSAT execution. This might have happened because the Blue Horizon machine became unavailable for scheduled maintenance. The total number of IPS was multiplied by more than five times when some batch jobs became active. It reached up to 110 *Billion* IPS.

Another measure of performance, is how much of the batch job maximum computational power is actually used by GridSAT processes. Most other parallel jobs run on all the processes from start to finish with little overhead. In this case, batch jobs are efficiently used. In the of case GridSAT, however, there are two main sources of inefficiency. First, some jobs might wait ideally at the start. Batch jobs usually include a large number of processes. Some of these processes have to wait until a sufficient number of splits occur to generate new sub-problems for all the newly created solvers. Second, some batch processes may contain idle solvers for a period of time after they solve the previously assigned sub-problem. The solver in this case, waits until it is assigned a new sub-problem by the master. For the first job in figure 3, which is a large 100-node job, the efficiency is 98.9%. Thus GridSAT was able to use batch jobs efficiently. The main reason is that batch jobs usually wait in the batch queue for a long time before executing. Thus by the time the job is executed, GridSAT was unable to solve the problem because it is hard. This means that batch jobs are only used when the problem is in deed hard. It is possible that for certain problems, the efficiency of batch jobs might be low. In this case, future versions of GridSAT might monitor the batch job efficiency to determine whether and when a job is to be terminated.

During our experiments, the Blue Horizon super-computer was being decommissioned. GridSAT was able to continue running experiments on the set of available resources through this transition. The scheduler would try to submit jobs but it would notice that the Blue Horizon resource was not responding. The failure of this single (but important)

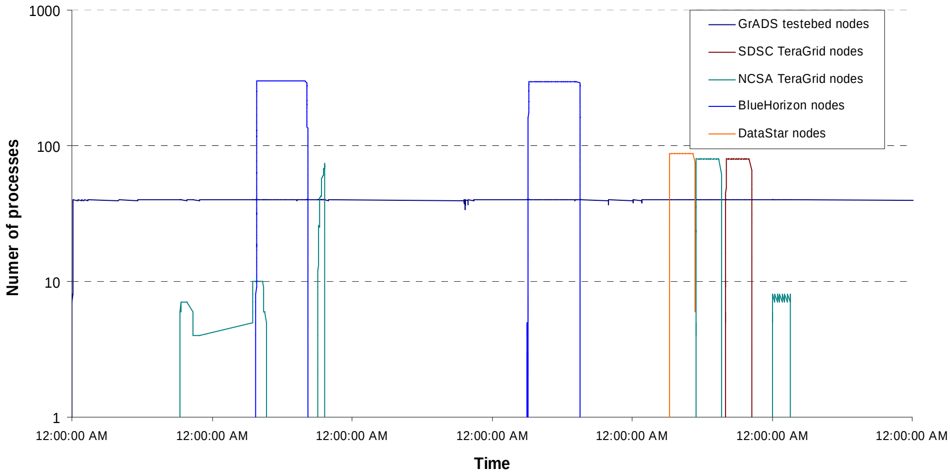

Figure 3. A six day snapshot representing GridSAT processor count usage from the different resources in logarithmic scale.

resource which did not affect the already running experiments shows the robustness of GridSAT.

6. Conclusion and Future Work

This paper presents a new version of GridSAT which implements a parallel, distributed and complete satisfiability solver. In order to solve harder problems, new improvements to both the algorithm and architecture of GridSAT were introduced. GridSAT is capable to dynamically selecting resources to enable improved overall performance.

We compared three methods for merging shared clauses used by GridSAT and showed that periodically merging clauses improves the solver the most. Also communication overhead is reduced by selectively sending important information first and avoiding redundancy when possible.

The experiments we presented show GridSAT's ability to manage and use a diverse set of dynamic computational grid resources. The experiments lasted for weeks as a testament to the robustness of the application. During these experiments new previously unsolved problems from practical and theoretical fields were solved.

As a result GridSAT represents one of the first examples of a true Computational Grid application. As "power" is added to the grid GridSAT can access, it makes efficient use of that power without undue waste. Its multi-infrastructure implementation using EveryWare makes it possible to incorporate the widest possible set of resources (i.e. non

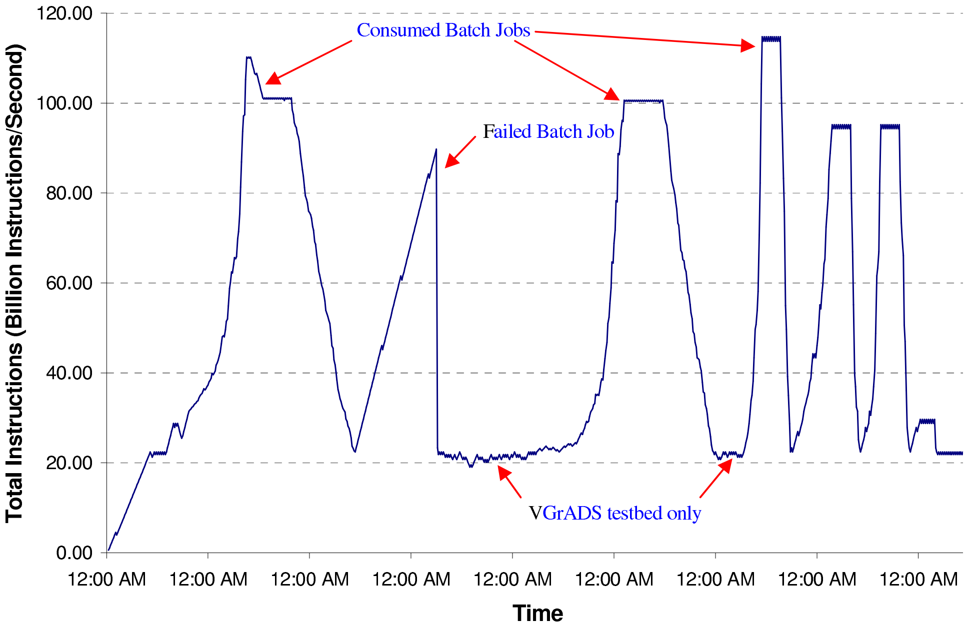

Figure 4. Estimation of Instructions per second usage for all resources during the same six day snapshot shown in figure 3.

are precluded because a particular infrastructure running at a particular version is not installed). Fluctuations in available power are tolerated automatically so that the overall application remains maximally efficient while it is executing. Finally, by solving previously unsolvable industrial problems, GridSAT has used grid computing in its true form to achieve new domain science.

References

Wahid Chrabakh and Rich Wolski. GridSAT: A chaff-based Distributed SAT solver for the Grid. Supercomputing Conference, Phoenix, AZ, November 2003. www.cs.ucsb.edu/ chrabakh/papers/gridsat.pdf.

Mathew Moskewicz, Conor Madigan, Yin Zhao, Lintao Zhang and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. 38th Design Automation Conference (DAC2001), Las Vegas, June 2001.

Ian Foster and Carl Kesselman, editors. The Grid – Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 1998.

Francine Berman, Geoffrey Fox, and Tony Hey, editors. Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons Inc., 2002.

Wilfred W. Li, Robert W. Byrnes, Jim Hayes, Adam Birnbaum, Vicente M. Reyes, Atif Shahab, Coleman Mosley, Dmitry Pekurovsky, Greg B. Quinn, Ilya N. Shindyalov, Henri Casanova, Larry Ang, Fran Berman, Peter W. Arzberger, Mark A. Miller and Philip E. Bourne. The encyclopedia of life project: grid software and deployment. New Generation Computing, volume 22-2, 2004, pages 127–136.

Gabrielle Allen, David Angulo, Ian Foster, Gerd Lanfermann, Chuang Liu, Thomas Radke, Ed Seidel and John Shalf. The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. The International Journal of High Performance Computing Applications,volume 15-4,pages 345–358, 2001.

J. P. M. Silva. Search Algorithms for Satisfiability Problems in Combinational Switching Circuits. Ph.D. Thesis, The University of Michigan, 1995.

G. Nam and F. Aloul and K. Sakallah and R. Rutenbar. A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints. International Symposium on Physical Design (ISPD), Sonoma Wine County, California, pages 222–227, 2001.

W. Kunz and D. Stoffel. Reasoning in Boolean Networks: Logic Synthesis and Verification Using Techniques. Kluwer Academic Publishers, Boston, 1997.

Tracy Larrabee. Test Pattern Generation Using Boolean Satisfiability. IEEE Transactions on Computer-Aided Design, pages 4–15, January 1992.

E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. Design, Automation, and Test in Europe (DATE '02), pages 142-149, March 2002.

E. A. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. PDMI preprint 9/2001, Steklov Institute of Mathematics at St.Petersburg, 2001.

Armin Biere. Limmat. http://www.inf.ethz.ch/personal/biere/projects/limmat/

JP M Silva and Karem Sakallah. Grasp - a new search algorithm for satisfiability. ICCAD. IEEE Computer Society Press, 1996.

C Sinz, W Blochinger and W Kuchlin. PaSAT - Parallel SAT-Checking with Lemma Exchange: Implementation and Applications. Proceedings of SAT2001, pages 212-217, 2001.

Sean L. Forman and Alberto M. Segre. NAGSAT: A Randomized, Complete, Parallel Solver for 3-SAT. Proceedings of SAT2002, 2002.

Bernard Jurkowiak, Chu Min Li and Gil Utard. Parallelizing Satz Using Dynamic Workload Balancing. Proceedings of Workshop on Theory and Applications of Satisfiability Testing (SAT'2001), pages 205–211, Boston, June 2001.

SAT Competition. http://www.satlive.org/SATCompetition/.

SAT Competition 2002, 2003. http://satlive.org/SATCompetition/.

Wahid Chrabakh and Rich Wolski. GrADSAT: A Parallel SAT Solver for the Grid. UCSB Technical Report, 2003-05, March 2003.

I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. IJSA, 1997.

Message Passing Interface Forum. MPI: A message-passing interface standard. University of Tennessee, Knoxville,CS-94-230,1994.

OASIS Web Services Reosurce Framework (WSRF) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.

The W3C Web Services Architecture working group, public draft, August 2003. http://www.w3.org/TR/2003/WD-ws-arch-20030808/

Steve Tuecke, Lily Liu and Sam Meder. http://docs.oasis-open.org/wsrf/2005/03/wsrf-WS-BaseFaults-1.2-draft-04.pdf.

Steve Graham and Bryan Murray. http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf.

M Davis, G Logeman and D Loveland. A machine program for theory proving. Communications of the ACM, Volume 5, pages 394–397, 1962.

M. H. Schulz and E. Auth. Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification. IEEE Transactions on ComputerAided Design, Volume 8, number 7, pages 811-816, July 1989.

T. Larrabee. Efficient Generation of Test Patterns Using Boolean Difference, pages 795-802, 1990.

K. Czajkowski, S. Fitzgerald, I. Foster and C. Kesselman. Grid Information Services for Distributed Resource Sharing. HPDC10, 2001.

Rich Wolski. Experiences with Predicting Resource Performance On-line in Computational Grid Settings. ACM SIGMETRICS Performance Evaluation Review, Volume 30, Number 4, pages 41–49, 2003.

Rich Wolski, Neil Spring and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. Future Generation Computer Systems, 1999.

Daniel Jackson and Mandana Vaziri. Finding Bugs with a Constraint Solver. International Symposium on Software Testing and Analysis, 2000.

Rich Wolski, John Brevik, Chandra Krintz, Graziano Obertelli, Neil Spring and Alan Su. Running EveryWare on the Computational Grid. SC99 Conference on High-performance Computing Proceedings, 1999.

Rich Wolski, John Brevik, Graziano Obertelli, Neil Spring and Alan Su. Writing Programs that Run EveryWare on the Computational Grid. IEEE Transactions on Parallel and Distributed Systems, volume 12, number 10, 2001. available from http://www.cs.ucsb.edu/ rich/publications/ev-results.ps.gz.

F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, D. Reed, L. Torczon and and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. International Journal of High Performance Computing Applications, Volume 15, Number 4, 2001. http://hipersoft.cs.rice.edu/grads/publications_reports.htm.

Ramn Bjar and Felip Many". Solving the Round Robin Problem Using Propositional Logic, AAAI/IAAI, pages 262-266, 2000.

B. Li, C. Wang and F. Somenzi. Abstraction Refinement in Symbolic Model Checking Using Satisfiability as the Only Decision Procedure. International Journal on Software Tools for Technology Transfer (STTT), December, 2003.

, Alessandro Armando and Luca Compagna. Abstraction-Driven SAT-based Analysis of Security Protocols. Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, pages 257-271, 2003.

Henry Kautz and Bart Selman. Planning as satisfiability. Proceedings of the Tenth European Conference on Artificial Intelligence, pages 359-379, August, 1992.

D. Jackson and M. Vaziri. Finding Bugs with a Constraint Solver. International Symposium on Software Testing and Analysis, 2000.

GrADS. http://hipersoft.cs.rice.edu/grads.

VGrADS. http://hipersoft.cs.rice.edu/vgrads.

The TeraGrid Home Page. http://www.teragrid.org

T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. Dr. Dobbs Journal, February 1995.