Accelerating Boolean Satisfiability through Application Specific Processing

Ying Zhao, Sharad Malik,
Department of Electrical Engineering, Princeton University, NJ

Matthew Moskewicz
Department of Electrical Engineering and Computer Science, University of California at Berkeley, CA

Conor Madigan
Department of Electrical Engineering and Computer Science, MIT, MA

Abstract
This paper presents our work in developing an application specific multiprocessor system for SAT, utilizing the most recent results such as the development of highly efficient sequential SAT algorithms, the emergence of commercial configurable processor cores and the rapid progress in IC manufacturing techniques. Based on an analysis of the basic SAT search algorithm, we propose a new parallel SAT algorithm that utilizes fine grain parallelism. This is then used to design a multiprocessor architecture in which each processing node consists of a processor and a communication assist node that deals with message processing. Each processor is an application specific processor built from a commercial configurable processor core. All the system configurations are determined based on the characteristics of SAT algorithms, and are supported by simulation results. While this hardware accelerator system does not change the inherent intractability of the SAT problems, it achieves a 30-60x speedup over and above the fastest known SAT solver -- Chaff. We believe that this system can be used to expand the practical applicability of SAT in all its application areas.

Keywords
Boolean satisfiability, Application specific, Multiprocessor, Configurable processor core

1. Introduction
SAT serves as the canonical NP-complete problem, and thus has received significant attention in the theoretical computer science community. It is also a practical problem encountered in several application domains, especially in Electronic Design Automation (EDA) and Artificial Intelligence. In the EDA domain, SAT is embedded in many areas of design synthesis and validation [1,2,3] - some of these being test pattern generation, combinatorial and sequential logic verification, logic synthesis, functional timing analysis, and routing. Any acceleration in solutions to this problem will have direct benefit for many applications in these areas.

Since the basic Davis-Putnam search algorithm [4] was proposed forty years ago, significant effort [5,6,7,8] has been spent in determining efficient heuristics that accelerate solutions for problem instances encountered in practice. Recently, we have seen the development of Chaff [8], a highly efficient algorithm that demonstrates 10-100x speedup compared to all previous software solutions. This has enhanced the scope of problem instances for which we can now find solutions in reasonable compute time using SAT software packages running on general-purpose processors.

Recently the development of high-density programmable logic has led to the development of a class of SAT accelerators by several research groups [9,10]. These accelerators exploit the fact that programmable logic enables a hardware accelerator to be synthesized specific to each *instance* of the SAT problem. These works have been able to demonstrate several orders of magnitude speedup for many problem instances. However, the hardwiring of the algorithms greatly compromises the ability to support complex control and dynamic data structures. Thus, these solutions cannot utilize the advanced features in the algorithms, which have been shown to bring orders of magnitude in speedup. In fact, these accelerators cannot tackle the really difficult problem instances that they were expected to deal with. Besides, the circuit compilation cost has in many cases become the dominant component of the solution time. It is exactly these limitations that we hope to overcome by using a programmable solution, but by using specially matched architectures rather than general-purpose processing. Such processors are referred to Application Specific Processors (ASPs) and are seeing increasing use in various application domains. Recently, some commercial offerings [11] of configurable processor cores have cut the development cost for ASPs significantly.

Developing parallel solutions [12,13] to deal with computationally intensive problems like SAT is not a new idea. However, all the current parallel SAT algorithms target general -purpose multiprocessor architectures and their associated operating system support, which have a very high communication overhead. Consequently, they are all based on the partitioning of the search space and utilize the coarse parallelism among searches in different subspaces, to avoid the high communication cost. Until now, the speedup and scalability of these algorithms have not been very encouraging, because they are likely to cause significant load-imbalance due to unpredictable workloads in the separate sub-spaces.

The multiprocessor system presented in this paper is built with specialized processing nodes and communication network. Each processing node consists of an ASP built from a commercial configurable processor core and a communication assist hardware unit for message passing. Through this system, we hope to

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

achieve significant speedup over current general purpose processing.

The rest of this paper is organized as follows. The SAT algorithms are introduced in next section. Section 3 presents the system prototype. The simulation environment is described in Section 4. Details of the performance evaluation of the system are presented in Section 5.

2. SAT Algorithms

Given a Boolean formula that is typically expressed in conjunctive normal form (CNF), the goal of SAT is to either find an assignment to the variables so that the formula evaluates to 1, or determine that no such assignment exists.

Although there are several different types of SAT algorithms, backtrack based complete search algorithms are the most popular. The high-level data flow shown in Figure 1 is applicable to all the algorithms falling into this category. They begin with an empty assignment. **decide()** selects a branching variable and assigns a 0 or 1 value to it. Then, **deduct()** performs BCP (Boolean Constraint Propagation) to determine the direct and transitive implications of this assignment. If no conflict is detected during **deduct()**, **decide()** is called again to pick the next assignment, and the procedure repeats. Otherwise, **diagnose()** is performed to determine the reason of the conflict, find the backtrack point, and generate the conflict clause. Then, **backtrack()** will remove the impact of all the implications becoming invalid and pick a new decision. In addition, **db_compact()** is another subroutine that has become very important while dealing with big problem instances encountered in practical applications. It performs clause deletion, clause database compaction and clause database updates/cleanup after the compaction, to maintain reasonable cache performance.

```
SAT_SOLVER() {
    while (true) {
      if (decide() == FALSE)
        return;                    //problem is satisifiable
      if (deduct() == CONFLICT) {
        if (diagnose() == FALSE)
          return;      //problem is unsatisfiable
        else
          backtrack();
      }
    }
```

Figure 1 Backtrack based SAT algorithm

In addition to the sequential algorithms introduced above, researchers have proposed various parallel SAT algorithms [12,13], trying to utilize the computation power of multiprocessor systems. In these algorithms the search space is partitioned by a control processor into small disjoint subspaces, each of which is assigned to a processor. As shown in
Figure 2, the processors search the assigned subspace using a sequential SAT algorithm. All the processor will return SAT if any of them finds a solution, otherwise UNSAT if there is no unexplored search space left. Since the search space is inherently unbalanced, all these algorithms use some type of dynamic load balancing techniques to achieve decent performance.

```
Parallel_SAT_Solver ()
{
   while (there are unexplored spaces) { //control processor
      for (i=1, …, N-1) {        //N-1 search processors
         if ( Sequential_SAT_Solver () == SAT )
            return SAT;
      }
   }
   return UNSAT;
}
```

Figure 2 Current, typical parallel SAT algorithms

3. System Prototype

There are always two tightly correlated elements involved in the application specific system design: the application and the underlying system architecture. The performance of the system is determined by how well these two elements match each other. Developing an efficient parallel system opens up a broad set of design considerations. Obviously, it is impractical to try all the possible combinations. Therefore, we start with an initial parallel algorithm and system architecture determined via application analysis. Then, the design process is an iterative refining procedure for both these two elements, supported by the simulation results.

3.1 Algorithm Characteristics

Among the most popular complete search algorithms, we have GRASP[5], SATO[6], Satz[7] and most recently Chaff[8]. These algorithms share the same high-level control flow and primitive operations, thus targeting these features is likely to be beneficial to future algorithms that may be developed.

We have done an extensive application analysis based on these algorithms by running them on a general-purpose single processor. The conclusion is that SAT problems are control, computation and data intensive. More specifically, they have the following features:

1. More than 99% of the execution time is spent on four major subroutines: **decide()**, **deduct()**, **backtrack()** and **db_compact()**.
2. There is a large amount of coarse and fine grain parallelism in the application.
3. SAT algorithms are all extremely memory bandwidth intensive, which suggests a distributed memory MIMD multiprocessor architecture.

3.2 A New Parallel Algorithm

Based on the algorithm characteristics, we propose a new parallel algorithm – MP_SAT. In terms of control flow, MP_SAT is similar to a sequential algorithm, with the computational intensive part being distributed. Unlike the existing parallel SAT algorithms, MP_SAT focuses on utilizing the fine grain parallelism in the clause and variable operations. In fact, utilizing fine grain parallelism is the basic idea behind most hardware accelerators. However, it usually implies higher communication cost for programmable solutions, and thus has been limited to software packages containing regular data accesses and operations, like those found in most signal and image processing

applications. In our design, we will show how to match the system architecture with the algorithm, so that we can exploit the potential for higher performance improvement and better processor load balancing associated with fine grain parallelism, and avoid the costly communication at the same time. In this section, we introduce the tasking partitioning and communication scheme in MP_SAT.

3.2.1 Task Partitioning

MP_SAT uses a hybrid task partitioning method that combines data based and function based decomposition. Each of the computationally expensive functions in all SAT algorithms repeatedly performs the same operations on a large set of data and there is no strong correlation among the data. This implies a natural data partition: each processor has a subset of the clauses and variables. And the functions running on each processor works on its own data subset in parallel, as shown in Figure 3. Through the data partitioning, not only multiple operations can be performed in parallel, in many cases, the cost of each individual operation is also reduced. These processors are called PPs (Processing Processors).

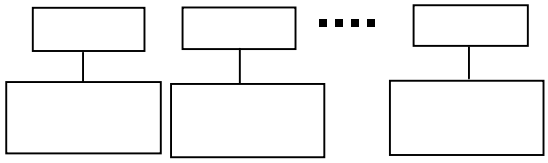

Figure 3 Data partition based parallelization

Other than the PPs, a Master Processor (MP) is needed to monitor the search state, coordinate the operations of all processors and perform global functions like **diagnose()**.

3.2.2 Communication and Synchronization Scheme

In a distributed memory MIMD system, processors exchange data via explicit message passing. Careful design of the communication scheme in both the algorithm and the hardware system is needed to minimize the communication cost.

In MP_SAT, the timely detection of new messages is not critical. For example, while a PP is doing **deduct()**, it need not know that new implications are coming in immediately since it can only start to process the new implications after it finishes the current one. In addition, as will be introduced in Section 3.3, the message buffers provided by the system can prevent the busy status of the message receiver stalling the sender. Thus, polling provides an efficient synchronization method with low overhead in our system, in contrast to interrupt. Each processor checks for new messages whenever it completes processing the current message. By doing this, we avoid wasting machine cycles caused by over polling and prevent the message buffers from being over loaded with unread messages at the same time.

There is one special message that is time sensitive: conflict. Although it does not affect the results of the program if the conflict message uses the same transmission scheme as other messages, it may affect the performance and complicate the control. Basically, all operations that a PP performs from the time MP detects a conflict until the time the PP starts to process the conflict messages are useless operations that will cause extra amount of work during **backtrack()**. Thus, a direct broadcast channel is used specifically for a conflict message. Global synchronization is also performed through the broadcast channel.

3.3 Hardware architecture

3.3.1 High level architecture

MP_SAT has two features: intensive communication based on message passing and message driven operations. Transmitting messages at high speed and with least interference with the computation are important considerations in the system design. We achieve these goals by using separate hardware units for communication, which greatly increases the parallelism between computation and communication. Therefore, each processing node in the system consists of a processor and a communication node. The processors have been designed to be straightforward message generator/consumer, with the corresponding communication nodes performing the message routing and buffering. Figure 4 shows the architecture with these processing nodes connected via a two-dimensional mesh.

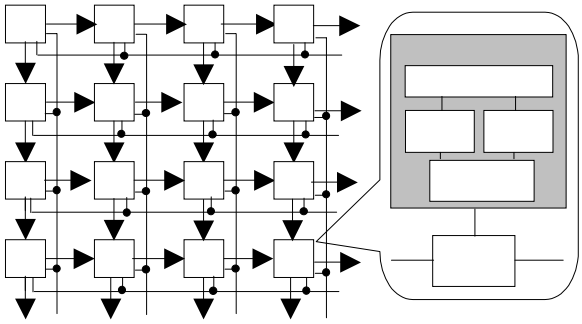

Figure 4 System architecture using embedded DRAM

The global broadcast wire is pipelined to avoid the delay and crosstalk problems associated with deep sub-micron designs. As shown in Figure 4, there is one pipelined broadcast line in each row and each column. A processor sends out the conflict message by setting the global wire to high. The signal is transferred via the vertical wire to each row, then via the horizontal wire to each processor. The maximum delay of this wire is 2n cycles, where n is the dimension of the network.

During our simulation, we have found that the parallel execution of communication and computation can bring more than 50% in overall performance, which shows that using separate hardware for communication is very efficient for communication intensive applications.

3.3.2 Processors

The processors in our system are application specific processors built from a commercial configurable processor core. These processors are customized to match the data access patterns and atomic operations in SAT algorithms. For details about the processor design, please see [14]. Here, we just give a short overview of them. To accommodate the memory bandwidth requirement, the datapath of the processor is chosen to be 128 bits. 11 new instructions are added to the basic configurable processor core, which fit in three categories: bit operation, parallel operations utilizing 128bits datapath and compound instructions for frequently appearing instruction sequences. The new

instructions bring about a 2-4X speedup, at a cost of 2400 gates. The final processor has 53.9k gates, and runs at 300MHz. These processors also have direct input/output ports, which allow them to talk to other components directly, without going through the ALU or the standard bus interface.

3.3.3 Communication Node

Each communication node consists of two parts:

- Communication channels. Each of the channels is a FIFO buffer dedicated to a pair of communicating processors, storing the messages between these two processors. The number of channels needed in each communication node is affected by the network topology. Based on simulation results, the size of each channel is determined to be 20 messages.
- Control logic that performs the message routing and channel empty/full detection, which can be modeled as a Finite State Machine. These finite state machines are very small and have been synthesized from a Verilog description to a circuit with approximately 100 gates.

Figure 5 shows the architecture of a communication node in a two-dimensional mesh topology.

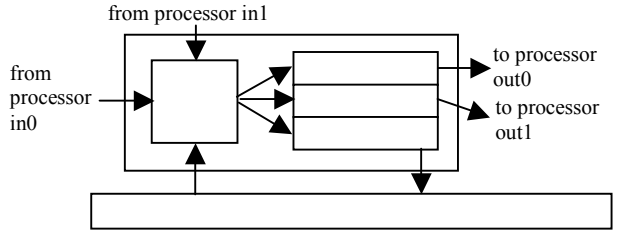

Figure 5 Communication node in a two-dimensional mesh

3.3.4 Communication Protocol

The communication protocol mainly includes the network topology and routing scheme. We chose a two-dimensional mesh topology due to the following reasons:

- Mesh topology gives a balanced bandwidth across all the processors. We have found that 90% of the messages in MP_SAT are broadcast messages. Thus, topologies like tree usually cause some "hot spots" that will become the communication bottleneck.
- Mesh has very good scalability and achieves good trade-off between the diameter and number of connections.
- Planar topologies like two-dimensional mesh make designing single chip multiprocessor system simpler, since all processors are connected with uniformly short connections.

In our system, the combination of message buffering and the parallel execution of computation and communication greatly reduce the possibility of network congestion. In addition, since all the operations in MP_SAT are message driven, transmitting messages fast is crucial in exploiting the parallelism. Therefore, we use deterministic routing. The communication nodes determine the route of a message based on the self-contained address information in its header.

To avoid the deadlock caused by loops in the network topology, a processor stalls whenever there are less than two spaces left in any of the buffers in its communication node. In this way, each communication node can guarantee that there is always empty space in all its local buffers. Thus, the messages can keep moving. Even if all the buffers are very close to full, as long as no new messages are dispatched to the network, the messages can be consumed and finally removed.

4. Simulation Environment

Our simulation environment consists of two simulators that run at different abstraction levels, accuracy and speed.

MP_SIM is a cycle-accurate multiprocessor simulator built based on a commercial cycle-accurate Instruction Set Simulator (ISS). It is designed to be highly flexible, so that can be used to model various communication schemes. It is also very easy to configure by making the network configuration and routing scheme transparent to user applications.

To overcome the speed limitation of cycle accurate simulation, we have implemented a C++ simulator that simulates the multiprocessor system behavior at a high level of abstraction. This simulator is event (message) driven. The cycle count is generated by calculating latencies for each individual operation (cache simulation, network communication, etc.). These latencies are calculated using the same architecture parameters. The purpose of this simulator is to get the overall cycle count, rather than to simulate the real cycle accurate behavior of the parallel algorithm. Since this simulator runs at a much higher level, it achieves approximately 50-80x speedup compared with the cycle accurate simulation, which allows us to simulate bigger problems running on more processors. First, we run the simulator on the examples we have run using MP_SIM, and compare the two sets of results. We find that the cycle counts obtained from the high level simulators are within [-12%, 5%] of the results from MP_SIM, which indicates that the results of the new simulator are reasonably close to the exact results.

5. Design Evaluation

5.1 Parallel Speedup

The speedup is the most basic metric to determine the validity of a parallel system design. Figure 6 shows the speedup of the parallel system while using different numbers of processors, compared with the basic Xtensa processor. The performance is measured using the number of clock cycles needed to solve the problems. The cycle counts of the parallel system are obtained using MP_SIM. The *sequential* results are obtained by running Chaff, a highly efficient sequential program on the single processor Xtensa ISS. Since they run at the same clock frequency, the *speedup* is calculated by dividing the cycle count of the parallel system by the *sequential* results. Generally speaking, the bigger the problem size, the higher speedup. This is understandable since larger problems usually have higher level of parallelism, and thus higher processor utilization rate.

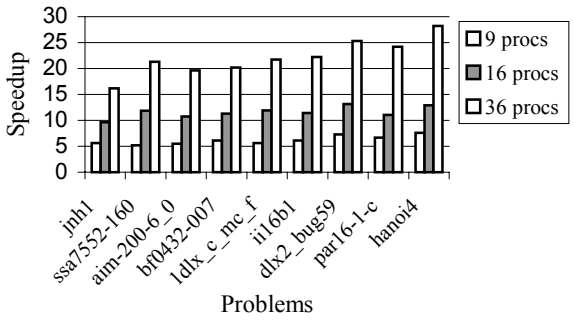

Figure 6 Speedup vs. number of processors

5.2 Scalability/efficiency Analysis

Figure 7 shows the efficiency of the parallel system, which is calculated by dividing speedup by the number of processors. Generally speaking, although speedup may increase, efficiency always decreases as the number of processors increases, due to the higher communication cost and the limited parallelism. This decrease is reflected very well in Figure 7. The figure also shows that the efficiency is generally higher for bigger problems.

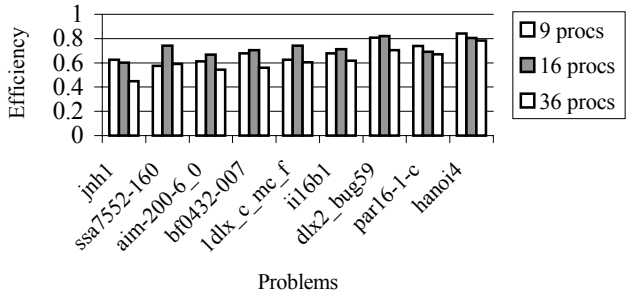

Figure 7 Efficiency vs. number of processors

For any parallel system, there must be a certain point where the efficiency has dropped so much that increasing processors may lead to lower speed. This is called the limitation of the scalability. This limitation is very important, since it also points out the performance limitation of the parallel system. We first perform a theoretical analysis of this limitation by examining how many operations can actually be performed in parallel. The parallelism of our system comes from three sources.

- Inserting variables to a sorted list in parallel in **backtrack()**.
- Compacting the clause database in parallel in **db_compact()**.
- Parallel execution of BCP in **deduct()**.

For almost all the problems we have experimented with, the number of variables to be processed per backtracking is less than or close to 100. Therefore, if we use more than 100 processors, there must be some processors remaining idle during **backtrack()**. Of course, the more processors we use, the smaller each sub-list, thus, the faster each insert operates. We expect that the parallelism limitation of **backtrack()** happens when the communication cost of collecting candidates from the PPs starts to outweigh the advantage of smaller list.

db_compact() has almost perfect parallelism as long as all the processors have approximately the same amount of clauses or literals. In addition, due to the data distribution, the data size on each processor in the multiprocessor system reduces to approximately 1/(number of processors) of the data size on a single processor. Thus, the need for database compact is greatly reduced. Actually, for all the problems we have run, none of them needs to call **db_compact()** when the number of processors is bigger than 16 in the parallel system. Therefore, for this function, more processor always means better performance.

The efficiency of parallel BCP is determined by the number of implications and clauses to evaluate upon one decision. In fact, even for the hardest problems we have seen so far, the number of clauses evaluated at each decision level is less than or close to 2000. We expect to see the parallelism in clause evaluation much smaller than this number since the generation of implications is a *chained* process. Actually, the average number of clauses to be evaluated per implication is only 5-10. Thus, the parallelism in BCP comes mostly from the parallel evaluation of more than one implication. This again confirms that the fast transmission of messages is very important for our algorithm. Since the parallel execution of BCP is the most important source of the performance improvement, it is also the main source of the limited parallelism.

The above analysis clearly demonstrates the limitation of the parallelism in MP_SAT. Figure 8 and Figure 9 show the speedup and efficiency of some big problems [15] running on the multiprocessor system, obtained using the high level simulator. These results show that our system can achieve approximately 80%-90% in efficiency for large size problems.

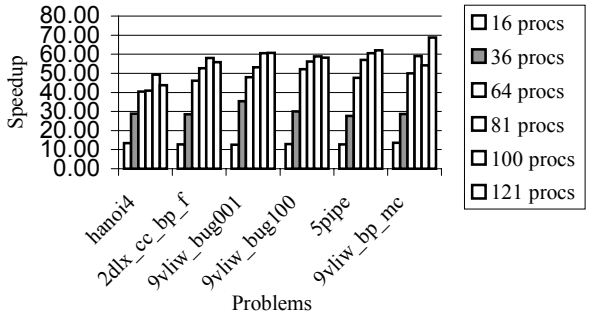

Figure 8 Efficiency vs. number of processors for big problems

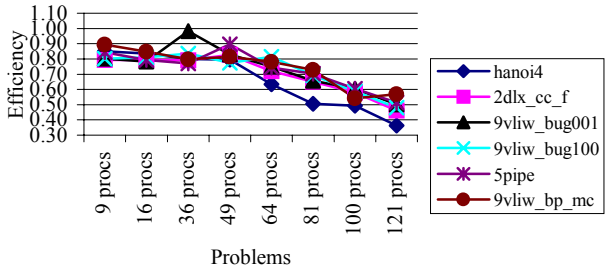

Figure 9 Efficiency vs. number of processors for big problems

From the above analysis and simulation, we can see that the limited degree of parallelism implies that this system is not scalable to very large number of processors. Although the optimum number of processors varies for different problem sizes,

even for the biggest problem in our set, a system with more than 81 processors will have significantly lower efficiency than a system with a smaller number of processors. Actually, during our simulation, we find that when the number of processors increases from 81 to 100, the system sees little performance improvement for most of the problems, if any. And we see slow down for most of the problem instances when the number of processors changes from 100 to 121. This is actually consistent with our theoretical analysis of the degree of parallelism available. We also see speed anomaly in some cases, caused by the difference between the search path of sequential and parallel algorithm. In many MIMD systems, the limitation of scalability is the ratio between communication and computation. However, in our system, the communication cost has been lowered significantly because of the usage of a separate communication node, the scalability is mainly limited by the amount of parallelism available.

5.3 Parallelism profile

The parallelism profile can be measured based on the utilization of each processor. The processor utilization is defined as:

utilization = busy_time/(busy_time+idle_time)

where, *busy_time* represents the time spent by the processor doing computation or communication, and *idle time* represents the time during which the processor was not doing anything. This includes the machine cycles while a processor is waiting for messages or is stalling because of the network congestion. Figure 10 shows the utilization of each processor in a 16-processor system. In general, we observe very high utilization of all the processors. Although it is not exact for each individual case, the utilization is generally higher for bigger problems. Besides, the load balancing between processors is very good, only varies by less than 2%.

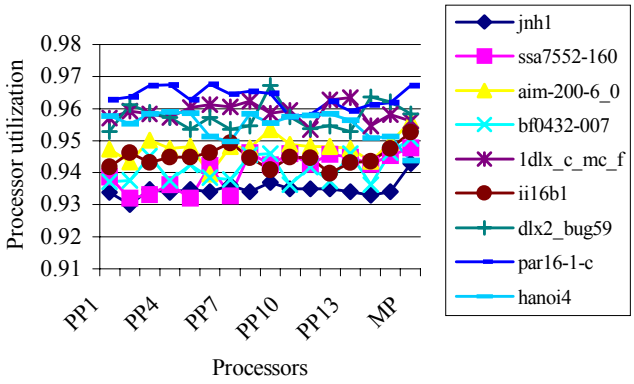

Figure 10 Parallelism profile of a 16-processor system

6. Conclusions

This paper has presented the design of an application specific multiprocessor system for SAT problems, which includes an application specific processor built from a commercial configurable processor core and a separate hardware communication unit on each processing node. The network topology and communication protocol are both determined based on the application characteristics. The system achieves a 30-60X over Chaff, the fastest known single processor SAT solver. We also show that very high efficiency can be achieved in a multiprocessor system, even for communication and control intensive algorithms. In the work introduced here, we utilize the most recent advances in several areas: high performance SAT solvers, commercial configurable processor cores and progress in the IC manufacturing techniques, to accelerate the SAT problems.

7. References

1 P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational Test Generation Using Satisfiability. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1167-1176, September 1996.

2 W. Kunz and D. Sotoffel. Reasoning in Boolean Networks. Kluwer Academic Publishers, 1997.

3 S. Devadas. Optimal Layout via Boolean Satisfiability. In *Proceedings of IEEE International Conference on Computer-Aided Design*, 1989.

4 M. Davis and H. Putnam. A computing Procedure for Quantification Theory. In *Journal of the ACM*, pages 201-215, 1960.

5 J. Silva and K. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *IEEE ACM International Conference on CAD-96*, page 220-227, November 1996.

6 H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272-275, July 1997.

7 C min Li. Equivalency Reasoning to solve a class of hard SAT problems. Available from http://www.cs.washington.edu/homes/kautz/challenge/cli.ps.

8 M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik. Engineering a (Super?) Efficient SAT solver. In *Proceedings of Design Automation Conference*, 2001.

9 M. Abramovici and D. Saab. Satisfiability on Reconfigurable Hardware. In *Seventh International Workshop on Field Programmable Logic and Applications*, September 1997.

10 P. Zhong, M. Martonosi, S. Malik and P. Ashar, Solving Boolean Satisfiabiligy with Dynamic Hardware Configuration. In *Proceedings International Workshop on Field Programmable Logic and Applications, FPL'98*, August 1998.

11 Xtensa configurable processor. http://www.tensilica.com

12 Benjamin W. Wah, Guo-Jie Li and Chee Fen Yu. Multiprocessing of combinational search problems. In *IEEE computer*. Pages 93-108, June 1985.

13 C. Powley, C. Ferguson and R. Korf. Parallel heuristic search: Two approaches. In *Vipin Kumar, P.S. Gopalakrishnan, and L.N.Kanal, editors, Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, NY 1990.

14 Y. Zhao, S. Malik, M. Moskewicz and C.Madigan. Matching Architecture to Application via Configurable Processors: A Case Study with Boolean Satisfiability. In *Proceedings of International Conference on Computer Design*, 2001.

15 Velev, M., FVP-UNSAT.1.0, FVP-UNSAT.2.0, VLIW-SAT.1.0, SSS-SAT.1.0, Superscalar Suite 1.0/1.0a, Available from: http://www.ece.cmu.edu/~mvelev.