A Distributed SAT Solver For Microcontroller

Tobias Schubert Bernd Becker

Institute for Computer Science
University of Freiburg, Germany
{schubert,becker}@informatik.uni-freiburg.de

Abstract: In this paper we present a parallel prover for the propositional satisfiability problem called PICHAFF. The algorithm is an adaption of the state-of-the-art solver CHAFF optimised for our scalable, dynamically reconfigurable multiprocessor system based on Microchip PIC microcontroller. Like usually in modern SAT solvers it includes lazy clause evaluation, conflict-driven learning, non-chronological backtracking, and clause deletion. A simple but efficient technique called Dynamic Search Space Partitioning is used for dividing the search space into disjoint portions to be treated in parallel by up to 9 processors.

Besides explaining of how such a complex algorithm could be implemented on simple microcontroller we also give experimental results demonstrating the potential of the implemented methods.

1 Introduction

The NP-complete problem of proving that a propositional Boolean formula is satisfiable (SAT) is one of the fundamental problems in computer science. Many problems can be transformed into a SAT problem in that way, that a solution of the SAT problem is also a solution of the corresponding original problem. On the basis of these facts a lot of developments in creating powerful SAT algorithms werde made in the last years: SATO [Zh97], GRASP [MSS96], or CHAFF [MMZ$^+$01] for example which are all based on the classical Davis-Putnam method introduced in the early 1960s [DP60, DLL62]. These algorithms have been successfully applied to real-world problems in the field of model checking, equivalence checking, or timing analysis to name only a few [BCC$^+$99, MSG99, SMSSS98].

Besides using faster CPUs parallel implementations seem to be a natural way to speed up SAT algorithms and by this to offer new opportunities in handling large problem instances. In the last decade powerful distributed SAT procedures have been developed: on one hand implementations for network clusters of general purpose workstations [ZBH96, SBK01, BSK03] and on the other hand realisations for special hardware architectures like transputersystems [BS96] or application specific multiprocessing systems [ZMMM01b].

In this paper we combine the different strategies by implementing a parallel version of a modern SAT solver for our special multiprocessor system using Mircochip microcontroller. The underlying hardware environment has been developed at the Chair of Computer Architecture in the last few years [BDB97, DDM$^+$00]. As the basis for this distributed SAT algorithm called PICHAFF we use one of the most competitive prover for the Boolean satisfiability problem: CHAFF. Hereby, all parts of the original implementation

have been optimised for the limited resources of the Microchip PIC microcontroller. The main aspects of our work are: (1) implementing a complex algorithm on simple microcontroller with restricted resources; (2) evaluating our multiprocessor system; and (3) getting experiences in this field, which might be helpful when realising a version of PICHAFF for a network cluster of standard PCs in the future.

The remainder of the paper is organised as follows: Section 2 introduces the SAT problem and the CHAFF algorithm. The multiprocessor system and the software environment are presented in Section 3. After that the implementation details and the parallel search method used in our approach are discussed in Section 4. Finally the results of the performance measurements are reported in Section 5 followed by a conclusion of the work done so far.

2 Satisfiability Problem

We assume the reader is familiar with propositional logic and related concepts. So we only give a short definition of the notations used throughout the paper:

- Let V be a set of n Boolean *variables*.
- $X = \{x, \overline{x} \mid x \in V\}$ is called the set of *literals*.
- $C = (x_{i_1} \vee \ldots \vee x_{i_k})$ with $x_{i_j} \in X$ is called a clause.
- $F = c_1 \wedge \ldots \wedge c_m$ with clauses c_i is called a formula in *conjunctive normal form* (CNF).

The propositional SAT problem now could be defined as the question if there exists an assignment for the variables in V such that the given CNF formula F gets satisfied.

As mentioned before most of the complete backtrack search SAT algorithms are based on the classical Davis-Putnam method. The pseudo-code not only of CHAFF but also of Berkmin, SATO, or GRASP for example which extend this concept with additional features is illustrated in Figure 1.

```
while(1) {
   if (decide_next_branch()) {
     while (deduce() == CONFLICT) {
       blevel = analyse_conflict();
       if (blevel == 0) return UNSAT;
       else back_track(blevel);
     }
   }
   else return SAT;
}
```

Figure 1: Pseudo-Code of modern SAT Solvers

The function `decide_next_branch()` selects the next branching variable. After that `deduce()` propagates the effect of the assigned decision variable: some clauses may become unit clauses. All the unit clauses are assigned to *TRUE* and the assignments are propagated until no further unit clauses exist. If all variables are assigned, a model for the given problem is found. Otherwise, if a conflict encounters the function `analyse_conflict()` is called: the reasons for the conflict are analysed and a backtrack level will be returned. The backtrack level indicates where the wrong decision was made and `back_track()` will undo all the wrong branches in order to preserve the conflict. A backtrack level of zero means that the given instance is unsatisfiable, because a conflict exists even without assigning at least one variable.

Nearly all modern complete SAT solvers are based on the algorithm described above, but they differ in the way the functions are implemented. In CHAFF the so-called *1UIP learning scheme* is used for conflict analysis stopping the process when the first *Unique Implication Point* (UIP) was found [ZMMM01a]. Intuitively, a UIP is the single reason that implies the actual conflict and has to be flipped to avoid the conflict.

For speeding up the `deduce()` function a lazy clause evaluation technique based on the notion of *watched literals* is used: depending on the value of these 2 watched literals it is easy to decide whether the clause is already solved (at least one literal defined correctly), a unit clause exists (one literal improperly defined), or a conflict occurs (both literals improperly defined).

There are several other features like the *Variable State Independent Decaying Sum* branching heuristic, clause deletion, and restarts integrated in CHAFF. Due to the page limitations, we will not delve into these topics, the reader may refer to [ZM03, MMZ$^+$01].

3 Multiprocessor System

In this section we describe the most important hardware and software components of our *Multiprocessor System* called MPS. For further information see also [BDB97, DDM$^+$00]. A picture of the layout is given in Figure 2. It mainly consists of three elements: the *Carrier Board* (CB), the *Processor Nodes* (PNs) and the *Communication Processor* (CP), that will be described below.

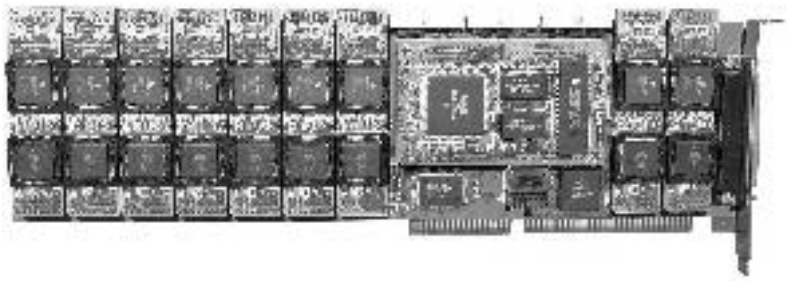

Figure 2: Multiprocessor System

3.1 Carrier Board (CB)

A long PC ISA slot card serves as the CB. Besides the communication processor up to 9 processor nodes fit onto one board. The CB is the core of the multiprocessor system and is used for *communication switching* between the different processors. Hereby, the connection between the PNs is established by a so-called Field Programmable Interconnection Device (FPID) from I-CUBE realising a hardware crossbar switch. Furthermore, all target applications, i.e. the SAT solver is downloaded via the CB into the external memory of the PNs using the interface to the PC. A dual port RAM on the CB serves for connecting the local bus of the ISA card to the PC bus.

3.2 Processor Node (PN)

The PNs are the basic computing units and consist of the following main characteristics:

- Microchip PIC17C43 RISC type CPU, 20 MHz operating speed
- 8 Bit data bus, 16 Bit address bus
- 454 Byte local RAM, 64 kWord external RAM, 4 kByte local ROM

The PIC17C43 microcontroller from Microchip was chosen for our purposes, since it is a low cost and for our applications satisfying module. The external RAM is reserved for the target applications of the PN, while the local ROM contains a simple operating system with basic functionality (refer to Section 3.4). The PN is equipped with one serial communication channel, capable of transferring data at 5 Mbit/s. The serial ports of all PNs are connected to the FPID device on the CB to enable communication between the PNs. This can be used to transfer SAT specific data like subproblems - as can be seen in Section 4.2 - or even to share useful information like generated conflict clauses for example.

3.3 Communication Processor (CP)

The CP - located on a separate board in the middle of Figure 2 - serves for handling the requests for communication issued by the PNs and for controlling the channel switching FPID on the CB. Some of the features are:

- Motorola MC68340 CISC type CPU, 16.78 MHz operating speed
- 32 Bit data and address bus
- 256 kByte RAM, 128 kByte ROM

In our approach the CP also handles the overall communication to the PC like downloading applications, transferring results and so on.

It is important to notice that the communication topology of the FPID can be reconfigured by the CP during runtime in less than 1 ms. Due to the fact that the crossbar switch

provides real hardware connections between the PNs the exchange of information can be done very fast and without the influence of the CP.

3.4 Software Environment

To use our multiprocessor system in a wide field of applications, the software framework is equally important as the hardware part. Therefore we implemented simple operating systems for the three main components - the PNs, the CP, and the PC - which are totally independent from the target application and provide the following functions: download of compiled program files (applications), configuration of the FPID module, exchange of data between the PNs, and transfer of received results.

Due to runtime critical aspects and the limited size of the internal ROM, the kernel for the PNs has been programmed completely in assembler. The operating systems for the CP and the graphical PC interface have been developed in *C/C++* using a *MC68340 Cross Compiler* and *Microsoft Visual C++ / NuMega DriverAgent* respectively (to access the carrier board via the PC ISA bus).

4 PICHAFF Implementation

In this section we present the realisation of PICHAFF for our multiprocessor system introduced in the section before. We will focus on the memory management, the data structures, the extensions needed for the parallel execution, and the overall application flow. Due to the limited resources of the Microchip microcontroller PICHAFF is also programmed completely in assembler.

4.1 Memory Management / Data Structures

A sketch of the data structures and the organisation of the 64 kWord external memory of the PNs is given in Figure 3.

At the top of the figure the overall partition of the memory is outlined. In our approach only the block ranging from address \$2000 to $\$FFFF$ (56 kWord) is available for the PICHAFF procedure and the given benchmark problem. This also limits the maximum size of the problem instances to approximately 4000-5000 clauses. To overcome this limitation an aggressive clause deletion mechanism has been integrated: if the number of clauses (initial ones and conflict clauses) exceed the available memory all conflict clauses that are currently *not active*[1] will be deleted. The reader should notice, that deleting non-active conflict clauses does not influence the correctness of the algorithm [ZM03]. Unfortunately, in the worst case it could happen that - even after the clause deletion process - there aren't enough memory cells left to store new clauses (for example if all assigned variables are implications forced by a *long* conflict clause containing a huge number of literals). In this case the algorithm stops with a corresponding failure message. As future work it would be interesting to analyse in more detail the trade-off between the loss

[1] In this sense an active clause currently forces an implication.

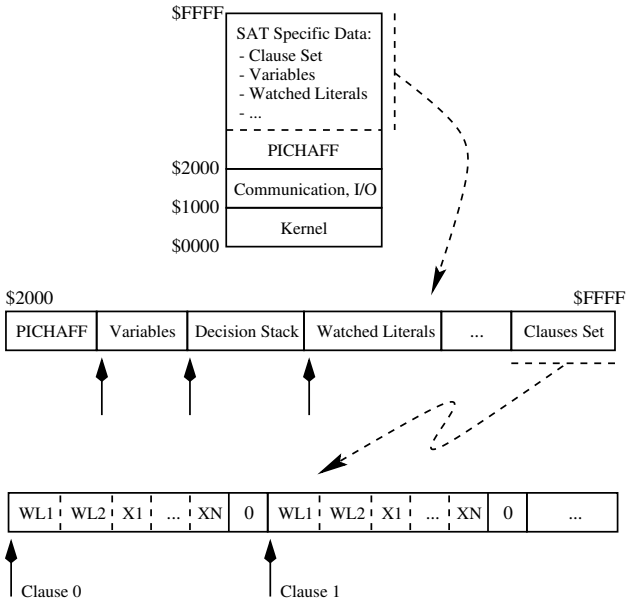

Figure 3: Memory Management

of information when deleting clauses and the benefit occuring from a decreased number of clauses, which normally results in a faster *Boolean Constraint Propagation* during the deduce step.

All parameters like the values of the variables, the decision stack, the lists of watched literals, or the clauses are arranged in a linear list. As can be seen in the middle of Figure 3 pointers are used to access the first element of each memory block.

In PICHAFF, the clauses follow the data structure given at the bottom of the figure: a pointer to the first literal of each clause and a special element (”0”) indicating the end of the clause. To avoid additional pointers and to have access in constant time the two watched literals for each clause are always located at the first two positions.

4.2 Parallel Search

In the previous sections we introduced our sequential PICHAFF procedure. For the parallel execution it has to be extended by a mechanism to divide the overall search space of the given benchmark problem into disjoint portions. These parts of the search space than could be treated in parallel by the processors.

To do so we adopt a technique called *Dynamic Search Space Partitioning* (DSSP) based on *Guiding Paths* (GP) [ZBH96, BSK03]. A guiding path is defined as a path in the search tree from the root to the current node, with additional information attached to the edges. Each entry in a GP consists of the following information: (1) the literal l_d selected at level d; and (2) a flag indicating whether the algorithm is in the first or second branch, i.e. if

backtracking might be needed at this point (Flag B) or not (Flag N).

Due to this definition it is clear, that every entry in a GP attached with Flag B is a potential candidate for a search space division, because the second branch has not been analysed yet. Thus the whole subtree rooted at the node of the search space corresponding to this entry in a GP can be examined by another processor.

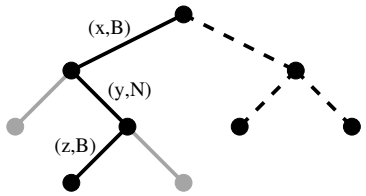

Figure 4: Guiding Path

An example for dividing the search space is given in Figure 4. Assume that the search process has reached the state indicated by the GP printed in black: $\{(x, B), (y, N), (z, B)\}$. A new task can be started by defining a second GP $\{(\overline{x}, N)\}$ (printed with dotted lines), as this part of the search space has not been examined so far. The original task will proceed the search after modifying its initial GP from $\{(x, B), (y, N), (z, B)\}$ into $\{(x, N), (y, N), (z, B)\}$ to guarantee that both processors work on different parts of the search space.

We have modified our PICHAFF algorithm to start at any arbitrary point in the search space by specifying an initial guiding path. This means that every time a PN gets idle during runtime, it contacts the CP by sending the corresponding signal. Then the CP opens a communication channel of the crossbar switch to another *active* processor, which is generating and encoding a *new* subproblem as a guiding path. This GP is finally transferred to the idle PN via the FPID device. An illustration of the communication process is shown in Figure 5. Notice, that the CP is only responsible for enabling and disabling the communication channels, not for transferring the data.

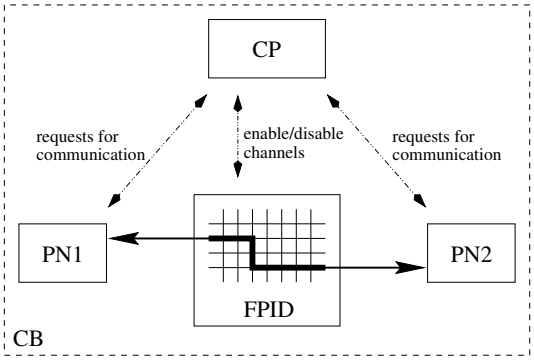

Figure 5: Principle of the Communication Process

4.3 Application Flow

Besides the implementation of the extended DP method for the PNs several supporting routines have been developed for the different types of processors, i.e. the CP and the PC. For this purpose we defined an application flow, which is carried out in 5 phases:

1. First of all the user specifies one or even more benchmark problems and the number of PNs.
2. Using the memory model given in Section 4.1 a problem specific PICHAFF algorithm will be compiled containing all the necessary SAT functions, the addresses of the memory blocks, the definition of the initial subproblems, the clauses, and the lists of watched literals.[2]

   For generating initial subproblems so-called splitting variables are assigned by the PC interface and sent to the PNs using a static variable order. This ordering of the variables is quite similar to the well known *Dynamic Largest Individual Sum* heuristic: as an example assume that 4 PNs should work in parallel and that the two literals with the highest occurrence in the given initial clause set are x_1 and $\overline{x_2}$. Then the guiding paths defining the four initial subproblems are $P_1 = \{x_1, \overline{x_2}\}$, $P_2 = \{x_1, x_2\}$, $P_3 = \{\overline{x_1}, \overline{x_2}\}$, and $P_4 = \{\overline{x_1}, x_2\}$.
3. The received assembler code of the previous step will be used as the target application and downloaded into the external memory of the PIC processors. After that all PNs get started.
4. If a PN gets idle the DSSP method presented in Section 4.2 is called.
5. After the search process is finished the results are sent to the PC and the next experiment could be started.

5 Experimental Results

For evaluating the performance of our implementation we made experiments using standard benchmarks available for download at `http://www.satex.org`. In columns 1 through 6 of Table 1 the main characteristics of the instances are given. The first column denotes the name of the benchmark set, where columns 2 through 4 denote the number of instances (#I), the number of satisfiable (#S), and the number of unsatisfiable benchmarks (#U). Also the number of variables (#V) and the number of clauses (#C) are listed there. The results for the parallel PICHAFF algorithm using 9 PNs (including *Dynamic Search Space Partitioning*) and for the sequential version using only 1 PN are presented in the last 3 columns of Table 1. The CPU times are always the sum of the CPU times needed to solve all the instances of the corresponding benchmark class.

As can be seen, the obtained speedup ranges from 4.2 to about 15 demonstrating that our methods work very well on a wide range of satisfiable and unsatisfiable benchmark problems. In case of the easily solvable benchmarks like the *jnh* instances, the overhead of

[2] Our experiments have shown that an optimised memory management according to the given benchmark results in a more compact assembler code of the PICHAFF algorithm and by this increases the performance.

Benchmark Set						1 PN	9 PN + DSSP	
Name	#I	#S	#U	#V	#C	CPU [s]	CPU [s]	Speedup
jnh	50	16	34	100	800-900	95.43	22.65	4.21
hfo3l	40	20	20	120	510	877.38	95.29	9.21
hfo4l	40	20	20	48	487	1111.29	101.25	10.98
hfo5l	40	20	20	29	621	799.74	77.17	10.36
hfo6l	40	20	20	21	924	624.19	76.10	8.20
hfo7l	40	20	20	16	1460	525.11	67.76	7.75
hfo8l	40	20	20	13	2325	673.28	85.63	7.86
hfokl	40	20	20	55	1087	477.84	70.79	6.75
uf125	100	100	0	125	538	1540.38	166.98	9.22
uuf125	100	0	100	125	538	4948.95	450.94	10.97
uf150	100	100	0	150	645	6041.10	470.26	12.85
uuf150	100	0	100	150	645	23953.10	1563.86	15.32

Table 1: Experimental Results

communication between the 9 processors is the most time consuming part and results in a lower speedup. In all other cases the received speedup is linear or even better. One reason for this *super-linear* behaviour is the fact, that the PNs explore different parts of the search space and by this usually generate different conflict clauses. These recorded clauses will not be deleted when a PN switches to a new subproblem. It obviously turns out that this obtained information is useful not only in the subproblem where the corresponding clauses have been created but also in the whole search space.

And secondly, the total number of clauses every processor has to deal with is smaller than the number of clauses one PN has to analyse in the sequential case resulting in a decreased number of clause deletion operations (necessary when a memory overflow has occurred) and an improved performance of the BCP procedure.

6 Conclusion

In this paper we demonstrated how a complex SAT procedure like CHAFF could be implemented using simple microcontroller. The PICHAFF algorithm has been developed in less than 2500 lines of assembler code. All features of modern backtrack search algorithms like lazy clause evaluation, conflict-driven learning, non-chronological backtracking, and clause deletion have been integrated and optimised for the limited resources of the Microchip PIC17C43 processors. For the parallel execution we enhanced PICHAFF by an efficient technique for dividing the search space using the FPID device of our multiprocessor system. A PC interface serves as an easy-to-use and comfortable back-end to configure all necessary parameters.

The experimental results point out the efficiency of the implemented methods and demonstrate the potential of our scalable, low cost multiprocessor system. Lastly, our experiences might also help to accelerate the performance of other classes of future parallel SAT solvers.

References

[BCC+99] Biere, A., Cimatti, A., Clarke, E., Fujita, M., and Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: *Design Automation Conference*. 1999.

[BDB97] Biermann, P., Drechsler, R., and Becker, B.: Modularity as key element in modern system design - a case study for industrial application of parallel processing. In: *European Design & Test Conference User Forum*. 1997.

[BS96] Böhm, M. and Speckenmeyer, E.: A fast parallel SAT-solver - efficient workload balancing. In: *Annals of Mathematics and Artificial Intelligence*. 1996.

[BSK03] Blochinger, W., Sinz, C., and Küchlin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. In: *Parallel Computing*. 2003.

[DDM+00] Drechsler, R., Drechsler, N., Mackensen, E., Schubert, T., and Becker, B.: Design reuse by modularity: A scalable dynamical (re)configurable multiprocessor system. In: *26th Euromicro Conference*. 2000.

[DLL62] Davis, M., Logemann, G., and Loveland, D.: A machine program for theorem proving. In: *Communications of the ACM*. 1962.

[DP60] Davis, M. and Putnam, H.: A computing procedure for quantification theory. In: *Journal of the ACM*. 1960.

[MMZ+01] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S.: Chaff: Engineering an efficient SAT solver. In: *Design Automation Conference*. 2001.

[MSG99] Marques-Silva, J. and Glass, T.: Combinational equivalence checking using satisfiability and recursive learning. In: *Design, Automation and Test in Europe*. 1999.

[MSS96] Marques-Silva, J. and Sakallah, K.: GRASP - a new search algorithm for satisfiability. In: *International Conference on CAD*. 1996.

[SBK01] Sinz, C., Blochinger, W., and Küchlin, W.: PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In: *Workshop on Theory and Applications of Satisfiability Testing*. 2001.

[SMSSS98] Silva, L., Marques-Silva, J., Silveira, L., and Sakallah, K.: Timing analysis using propositional satisfiability. In: *IEEE International Conference on Electronics, Circuits and Systems*. 1998.

[ZBH96] Zhang, H., Bonacina, M., and Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. In: *Journal of Symbolic Computation*. 1996.

[Zh97] Zhang, H.: SATO: An efficient propositional prover. In: *International Conference on Automated Deduction*. 1997.

[ZM03] Zhang, L. and Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: *Design, Automation, and Test in Europe*. 2003.

[ZMMM01a] Zhang, L., Madigan, C., Moskewicz, M., and Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: *International Conference on CAD*. 2001.

[ZMMM01b] Zhao, Y., Malik, S., Moskewicz, M., and Madigan, C.: Accelerating Boolean satisfiability through application specific processing. In: *14th International Symposium on System Synthesis*. 2001.