10th International Conference on Field Programmable Logic and Applications, August 2000.

A Parallel Pipelined SAT Solver for FPGA's *

Mark Redekopp and Andreas Dandalis

University of Southern California, Los Angeles CA 90089, USA
{redekopp, dandalis}@halcyon.usc.edu

Abstract. Solving Boolean satisfiability problems in reconfigurable hardware is an area of great research interest. Originally, reconfigurable hardware was used to map each problem instance and thus exploit maximum parallelism in evaluation of variable assignments. However, techniques to greatly reduce the search space require dynamic reconfiguration, and make regular mappings more desirable. Unfortunately, using a regular mapping constrains the parallelism in assignment evaluation. The architectures that have emerged choose either custom mapping and maximum parallelism or regular mapping and the promise of significant decreases in the search space. We propose a framework that can exploit both. Our framework uses a regular mapping while introducing a scalable parallel architecture. Using our approach, speedups of up to one order of magnitude over current state-of-the-art reconfigurable hardware solvers have been obtained.

1 Problem

Boolean satisfiability (SAT) is a well-known NP-Complete problem that seeks to find an assignment to a set of boolean variables given a set of clauses as constraints. One way to represent SAT is using a binary decision tree (BDT) with each level corresponding to a decision and each node corresponding to a particular set of variable assignments. Decisions are made by assigning variables a specific value. Each new decision is checked for consistency with the set of clauses. During this process implications occur based on clause constraints. Backtracking occurs if a certain decision leads to a conflict in any clause. Backtracking is the process of unassigning or reassigning variables that were previously decided. A problem is satisfiable if there exists a node in the BDT that satisfies all clauses.

There are two main options to speedup standard backtrack search algorithms for solving SAT. The first is to decrease the evaluation time of each node in the search tree. This is easily done in hardware where all clauses can evaluate an assignment in parallel. The other option is to decrease the number of nodes visited in the search tree via sophisticated backtracking techniques and adding clauses dynamically [1]. FPGA implementations usually choose to exploit one option or the other. Choosing to decreasing the node evaluation time led to architectures where all clauses were evaluated in parallel using an instance specific mapping of variables to clause circuits, as in [2]and [4]. However, to decrease the number

* This research was performed as part of the MAARCII project. This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-99-1-0004 monitored by Fort Huachuca

of nodes visited requires dynamically adding clauses which is very time consuming for instance specific mappings. Instead a pipelined ring of clause circuits, or modules, with variables passing through the pipeline was proposed in [3] because it allowed fast dynamic reconfiguration times for new clauses by simply adding a new module to the end of the pipeline.

The main advantage of implementing SAT in hardware is the decrease in unit propagation time. Unit propagation time is defined as the time it takes for a decision to be checked and either move on to the next decision or backtrack. This corresponds to evaluating a node or nodes in the search tree. The main portion of this time is spent while clauses make implications or raise conflicts. Hardware solvers let clauses generate implications and conflicts in parallel, greatly reducing the unit propagation time. An instance specific mapping yields a propagation time proportional to the number of transitive implications, t, (the chain of implications that may result from implying or deciding a variable's value) made from a decision. A pipelined mapping yields unit propagation time proportional to $t \cdot e$, where t is again the number of transitive implications and e is the number of clauses.

2 Our Approach

2.1 Overview

Our design introduces a framework for fast node evaluation and dynamic learning to reduce the number of nodes visited. We propose a tightly integrated, parallel pipelined architecture for fast node evaluation while maintaining a regular pipelined mapping to simplify dynamic reconfiguration (see Figure 1). Dynamic learning requires maintaining data structures of information learned from earlier processing. Similar to [3], we envision FPGA's interacting with the host machine via a run-time assist running on the host to maintain and process the data structures for dynamic learning. Then using self-reconfiguration techniques, dynamic learning logic on the FPGA itself could add clauses dynamically. This remains a key part of our future work with the parallel architecture for fast node evaluation being the primary contribution of this work.

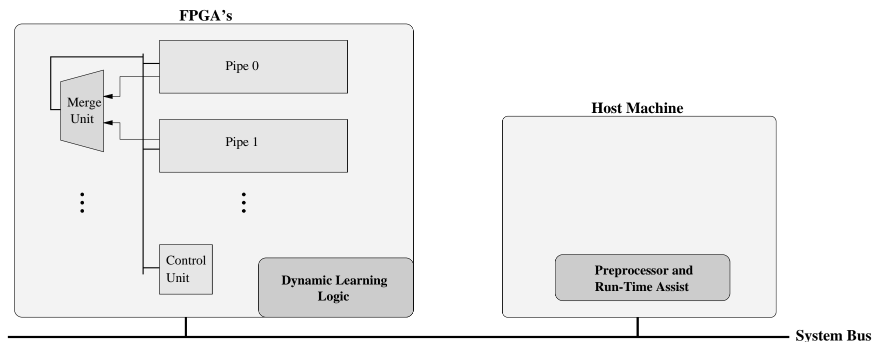

Fig. 1. Parallel Pipeline Framework

2.2 Parallel Pipelined Solver

Using a pipeline of clauses as our building block we split clauses into a set of parallel pipelines each with a set of variables cycling through the pipe and being merged when needed. For every decision or backtrack we run these pipelines separately and then merge their results until a consistent, global variable assignment is found. Our design of an efficient merging structure allows for tight coupling between pipelines and a high degree of parallelism. Since pipelines and merge units are regular in layout, fast compile times, fast clock speeds, and dynamic learning are possible. This design adds scalable parallelism by utilizing concepts similar to data parallelism, where multiple sets of data operate independently and are synchronized at certain points.

Given that a single pipeline has a unit propagation time of $t \cdot e$, our design decreases unit propagation time by shortening the pipeline. But, this action increases t since implications that could be found in one pass through the single pipeline may not be found by one pass through multiple pipelines. Also, creating independent sets of variables forces us to merge the results of the pipelines to update the implications or conflicts made in one pipeline to all the others before the pipelines can cycle again. This merge time must also be accounted for. Our unit propagation time is thus based on the number and cost of both merging and propagating variables through the pipelines. The unit propagation time of our design is:

$$UPT = t_p \cdot (\frac{e}{p} + \frac{v}{B}) + m_p \cdot (\log p + \frac{v}{B}) \qquad (1)$$

where t_p is the average number of iterations per decision for p pipelines, e is the number of clauses, v is the number of variables, m_p is the number of merges per iteration, and B is the bus width in terms of variables.

Architecture The basic architecture consists of a set of pipelines, operating in parallel. Each pipeline is similar in structure as that of current pipelined SAT solvers, such as the one in [3]. A single pipeline can be seen as a ring of clause modules pipelined together with a variable memory unit that cycles the variables through the clause modules. Clause modules are instance specific blocks that are based on a clause from the given SAT instance. Clause modules take a set of variables as input, generate implications or conflicts based on the input set, and produce an output set in a single cycle. Pipelines also have a variable memory that contains each variable's current value (0, 1, Undecided, or Conflicting). The variable memory is also responsible for implementing the state machine based on the given algorithm.

Fundamental to a tightly coupled approach is combining the information (implications or conflicts) from the pipelines into a consistent set of variable values. To do this we introduce a hardware block of merge units. Merging must be efficient to preserve the speedup gained from splitting the clauses into parallel pipelines. Therefore, we choose a tree structure of merging units. A subset of variables can thus be merged into their global state in $\log p$ steps. The actual merge hardware is simple as well. Using the standard two bit encoding of variables we can merge two variables through a simple bitwise OR'ing.

Algorithm The basic algorithm must now be modified to accommodate multiple pipelines. For the problem to be proven satisfiable, a variable assignment must be found that satisfies all pipelines. Thus, we use the merge unit to assure the same variable assignment is being checked for all the pipelines. Our approach performs this merging during every decision so that the pipelines will update implications and conflicts to each other. When a decision or backtrack is made, each pipeline is at the same node and will thus make the same decision or backtrack. Using this technique also assures that the same decisions and backtracks will be made as in the standard single pipeline implementation since all implications will eventually be communicated to each pipeline. The basic algorithm for evaluating a new decision is as follows.

1. All pipelines make the same decision on their own set of variables.
2. Variables are cycled through the pipeline until all implications from that pipe are found.
3. If a conflict is found at anytime, the pipe informs the others and each pipe backtracks and the process starts again.
4. If no conflicts are found and all the pipelines have finished, a merge is performed. If the merge causes a conflict, all pipes will backtrack. If the merge caused a variable to be changed (e.g. Pipe 1 has v = 0 and Pipe 2 has v = U) the new set must be updated to all pipelines and they will now iterate with this new set by going back to step 2. If however no variables were changed in the process of merging, a consistent set has been found and a new decision can be made after updating each variable memory with this new set.

With this architecture and given the fact that we will make the same decisions and backtracks as the single pipeline we can now arrive at an approximate speedup equation by simply comparing unit propagation times:

$$Speedup = \frac{UPT_1}{UPT_p} = \frac{t_1 \cdot (e + \frac{v}{B})}{t_p \cdot (\frac{e}{p} + \frac{v}{B}) + m_p \cdot (\log p + \frac{v}{B})} \tag{2}$$

Using this equation we can predict an optimal number of pipelines for speedup or area tradeoffs. The problem however is that t_1, t_p and m_p are intrinsic to the instance, the number of pipelines, and the partitioning of clauses. To find their exact values would require actually solving the problem, but if estimates can be found without running the entire problem, the equation can be used. Our future work will be to develop heuristics to find and use these estimates, and use self-reconfiguration to perform dynamic reconfigurations based on these parameters.

Another issue that needs to be addressed is the increase in area due to multiple pipelines. The number of clause modules is the same so the same area would be required in both our architecture and single pipeline architecture. The additional area comes from the variable memories and the merge units. If we have p pipelines the increase in area will be $p - 1$ variable memories each consisting of $4 \cdot v$ bit registers (2 sets of 2 bits for each variable) and some control logic to implement the state machine. Each merge unit is made of B (B = bus width) OR gates. For a tree structure there would be $\frac{p}{2} - 1$ such units. This increase in area is not prohibitive and thus allows for a reasonable number of pipelines.

3 Experimental Results

Using a C++ simulator that models a single pipeline and our parallel pipelined design to count cycles, we simulate both designs to solve specific instances. Using varying numbers of pipelines our parallel pipelined design showed speedups compared to the single pipeline. We have chosen problems from the DIMACS challenge set [5] and obtained the results shown in Table 1. The results show that moderate speedups can be obtained with even a few pipelines.

	Simulated Speedup *(No. of Pipelines)*					
Problem	**2**	**4**	**8**	**16**	**32**	**64**
par8-1-c	.94	1.32	2.21	2.64	2.99	3.21
hole6	1.23	2.08	3.32	4.69	5.88	6.73
aim-50-2_0-no-4	1.17	1.92	3.08	3.74	4.73	4.54
aim-50-1_6-yes1-1	1.32	1.88	2.96	3.67	3.73	3.34
aim-100-3_4-yes1-4	1.27	2.12	3.58	5.75	8.35	9.97

Table 1. Simulated Speedup over Single Pipeline Design

The nature of speedups can be characterized by accounting for two factors, the number of iterations per decision to find all implications and the number of merges per decision. Using equation (1) we see that when the number of pipelines is small the first term will dominate (i.e. most of the time is spent cycling the variable through the pipeline). This is the case with par8-1-c running with 2 pipelines. As the number of pipelines increases the merging time becomes the dominant term. This is the major reason why the speedups taper off after a certain number of pipelines. Also, as the pipeline length decreases to the point where it is always filled with variables, an increase in the number of pipelines will hurt performance (e.g. aim-50-1_6-yes1-1). Thus, if we can decrease the number of merges per decision, speedup will increase. It is also clear now that decreasing the number of iterations or passes per decision of the pipelines will yield greater speedups. This is difficult because the number of iterations per decision is primarily an intrinsic function of the instance itself. However, clause partitioning and ordering for each pipeline can make a difference. Unfortunately, to find an optimal ordering and partitioning requires solving the SAT instance. Alternately, certain heuristics could yield greater performance.

We also compared the speedups obtained from our simulator with the speedups predicted from equation (2). In all cases the predicted speedup was within a few percent of the actual speedup, but to find the intrinsic parameters t_1, t_p and m_p the simulator had to be run. However, estimates were found by running our simulator for a small number cycles. The resulting values for the parameters yielded at most 14% error after running only 20,000 cycles (less than 2% of the cycles needed to solve most of the above instances). This result shows great promise to finding heuristics to choose an optimal number of pipelines. It also opens up new areas of research for dynamic reconfiguration during runtime. As the parameters change, pipeline configurations could be changed during runtime to achieve optimal speedup. This remains part of our future work.

4 Future Work

Several key areas of our framework need to be explored. First and most important is a dynamic learning mechanism that can be integrated into our current framework. We thus plan to incorporate dynamic learning into our architecture and implement it using self-reconfiguration. Also researching possible heuristics for clause partitioning and optimal numbers of pipelines. Using learning techniques at runtime we plan to investigate the use of self-reconfiguration for increasing or decreasing the number of pipelines and better partitioning of clauses.

5 Conclusion

In conclusion, a new framework that exploits both fast node evaluation and regular mapping has been outlined. This parallel architecture and algorithm is unique in both hardware and software in that it uses a mechanism for tightly coupled data parallelism to solve SAT. We increased the amount of parallelism a pipelined design can attain by introducing multiple pipelines and developing efficient merge units to combine intermediate results. This yielded speedups of up to one order of magnitude over current state-of-the-art. Finally, we have presented a design that exploits parallelism and is still amenable to dynamic reconfiguration for dynamic learning.

The work reported here is part of the USC MAARCII project (http://maarcII.usc.edu). This project is developing novel mapping techniques to exploit dynamic reconfiguration and facilitate run-time mapping using configurable computing devices and architectures. Computational models and algorithmic techniques based on these models are being developed to exploit self-reconfiguration using FPGAs. Moreover, a domain-specific mapping approach is being developed to support instance-dependent mapping. Finally, the idea of "active" libraries is exploited to develop a framework for automatic dynamic reconfiguration.

6 Acknowledgments

A special thanks is extended to my advisor, Professor V. K. Prasanna for his leading and vision in this area.

References

1. J.M. Silva, *"GRASP - A New Search Algorithm for Satisfiability,"* Proc. Intn'l. Conf. on CAD, pp. 220-227, November 1996
2. P. Zhong, M. Martonosi, et al., *"Accelerating Boolean Satisfiability with Configurable Hardware,"* Proc. Symp. on Field-Programmable Custom Computing Machines, April 1998
3. P. Zhong, M. Martonosi, et al., *"Solving Boolean Satisfiability with Dynamic Hardware Configurations,"* Proc. Intn'l. Workshop on Field-Programmable Logic and Applications, Sept. 1998
4. Abramovici and Sousa, *"A SAT Solver Using Reconfigurable Hardware and Virtual Logic,"* Journal of Automated Reasoning, Vol 24, nos 1-2, pp. 5-36, Febr. 2000
5. DIMACS. Dimacs challenge benchmarks. Available at ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf