VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

Reverse Engineering of Gate-Level Netlists

Term Project
CS5234: Advanced Parallel Computation

Sonal Pinto
M.Eng. Computer Engineering
Department of Electrical and Computer Engineering
spinto64@vt.edu

CONTENTS

1 INTRODUCTION

Security of hardware Intellectual Property (IP) is a concern in the chip-design industry, where with sufficient technological resources such as electron microscopy and Advanced ATPG, it is possible for a determined entity in possession of the chip, to *steal* the design. The standard safety guard against such malicious activity is usually to obfuscate the physical design (say, by using redundant logic, bus codecs or propagation of logical *don't-cares*). This would certainly make the task of reverse engineering arduous and time-consuming, but not impossible.

Techniques to extract the high-level design information and architecture have been proposed by Wasson et al [1] and Subramanyan et al [2], using behavioral pattern mining and functional analysis respectively. These methods detail out much of the circuit, and allow for re-synthesizing the target design using the extracted information. But, in doing so, often lose the opportunity to reuse the physical design that has already been implemented. Hence, this project proposes a solution to completely resolve the design at the very gate-level, allowing the reverse engineering to reuse the hardware physical implementation without spending resources towards re-synthesis and subsequent planning and routing. For this project, we consider that most of the design has been extracted using the previously mentioned techniques, and that possibly less than 10% of the design remains unknown. These unknown gates shall be termed, `mystery gates` (MG).

Problem Statement: Given a gate-level netlist with Mystery Gates, and a simulation trace, the goal of the project is to find *all* possible solutions in terms of gates, that would match the hardware specification of the Mystery Gates (IO width), and completely satisfy the given simulation trace.

The project begins by describing the complexity of problem and the solution search space for a given problem size (number of mystery gates). The brute-force approach (`mgs_bf`) is contrasted against a novel solution, Functional Satisfiability (`mgs_sat`), where the problem is portrayed as a Maximal SAT problem (i.e. to find every possible solution, rather than just one, implying an exhaustive exploration). Then, the general algorithm and parallel implementation is explained. The project is developed on the shared-memory parallel computation paradigm, using OpenMP. Finally, evaluation is reported for circuits from the ISCAS'89 benchmark.

Keywords: Reverse Engineering, Electronic Design Automation, gate-level obfuscation

2 BACKGROUND

This section begins with describing the Brute-Force approach, and then the basic principles behind the Functional Satisfiability algorithm.

2.1 BRUTE-FORCE

A naive approach to solving the mystery gate problem is to apply every single permutation of the possible gates that could fit the IO (Input/Output) configuration for that gate. This is analogous to the brute-force attempt for discovering passwords in cryptography problems, and is similarly impractical.

For example, assume that a netlist with a single 2-input mystery gate is given. That gate could have 6 possible solutions (as per the gate-library on hand): `{and, nand, or, nor, xor, xnor}`. To resolve the correct gate identity, every possibility would have to be assigned to the mystery gate and the circuit would need to be simulated as per the given testbench (simulation trace). At every single input vector (a testbench is a list of input vectors), the overall circuit simulation output (`reference`) is checked against the expected (`golden`) output. Whenever it fails to match, the permutation is discarded and the next mystery gate possibility is assigned and simulated from the beginning. A solution, would exhaust the testbench without any error between the reference and golden output.

Note, that for N mystery gates of the same type as the above example, the permuted search space is 6^N. It is easy to realize that this is clearly not scalable. The brute-force algorithm (`mgs_bf` : *Mystery Gate Solver - Brute Force*) has been implemented for the sake of contrast. For circuit, c432 (204 gates), for a random input simulation of length, 128, and a problem size of 8 mysery gates, `mgs_bf` took 1 hour to find maximal solutions.

2.2 FUNCTIONAL SATISFIABILITY : BASIC PRINCIPLE

A Boolean logical gate, at any point of time in its functional execution can only output, either a logical High (`1`) or a logical Low (`0`). Therefore, considering that every mystery gate is indeed a Boolean logical gate, then for a simulation trace of length, K, the length of the execution state trace of a gate is 2^K, where at each input vector (excitation at the *Primary Inputs* of the circuit), the gate either outputs a `0` or a `1`, such that the golden circuit output matches the reference.

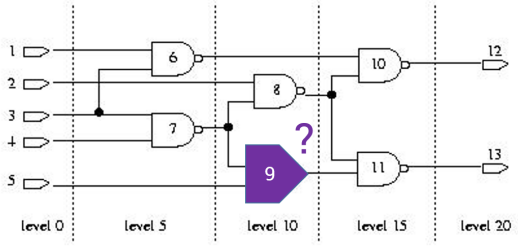


Figure 2.1: 2-input mystery gate at net#9 in c17

A solution *functionally satisfies* the circuit, then its execution trace perfectly simulates the circuit.

Key definitions and the principle will be illustrated with the following example. Consider the small circuit, c17 (Figure 2.1), with a 2-input mystery gate at net#9. From before, we know that a gate has a finite set of possibilities that are build from the gate-library on hand, that matches the IO configuration of that gate. This list is termed as the *suspect_list* or a *clause*.

Initially the *clause* for gate#9 is `{and, nand, or, nor, xor, xnor}`. At every input vector, we introduce a *Stuck-at fault* at output pin of the gate, thereby fixing the output to the fault (either `0` or `1`). The circuit is simulated and only the correct execution of the gate is recorded. For a correct execution, the inputs at the gate are noted and is used to reduce the *clause* (similar to interrogating the suspects). Furthermore, for executions, where both *faults* produce a correct circuit output, no reduction is possible, and the gate state is treated as a *don't-care*. Such cases appear when the input vector produces a *Observability Don't Care* at the mystery gate, render it's fanout non-dominant. Eventually, exhaustively simulating and recording the distinguishing *faults*, reduces the *clause* to a *unit clause*, where there is only suspect left in the list. This is marked as the solution for that mystery gate.

The illustrative execution of this principle for c17 is demonstrated in Figure 2.2. It can be noted that the clause for the mystery gate converges within two input vectors. In general, the input search space (execution trace) for N mystery gates, is 2^{NK}. Though, the actual search is only committed until convergence, as seen in the example. Once, we have unit clauses for all mystery gates, we enter the *verification phase*, i.e. we no longer need to introduce faults, and can simply verify the solution by running the testbench until exhaustion of input vectors. This relies on the fact that upon converging to a unit clause, any further simulation of this solution is expected to perfectly match the golden output. Upon failing this verification, the solution is discarded. Therefore, any converged solution can only be accepted if it exhausts the testbench.

```
Initial Clause: { and, nand, or, nor, xor, xnor }
-------------------------------------------------
Input Vector #1: 10001, Golden Output: 00
    Reference Output,
        Stuck-At-0 (gate#9): 01
        Stuck-At-1 (gate#9): 00   (valid)

    Reduction,
        gate input: 11, gate output: 1
        Clause = { and, or, xnor }
-------------------------------------------------
Input Vector #2: 00100, Golden Output: 00
    Reference Output,
        Stuck-At-0 (gate#9): 01
        Stuck-At-1 (gate#9): 00   (valid)

    Reduction,
        gate input: 10, gate output: 1
        Clause = { or }
-------------------------------------------------
Converged. Mystery Gate#9 = OR
```

Figure 2.2: Solving c17 using Functional SAT

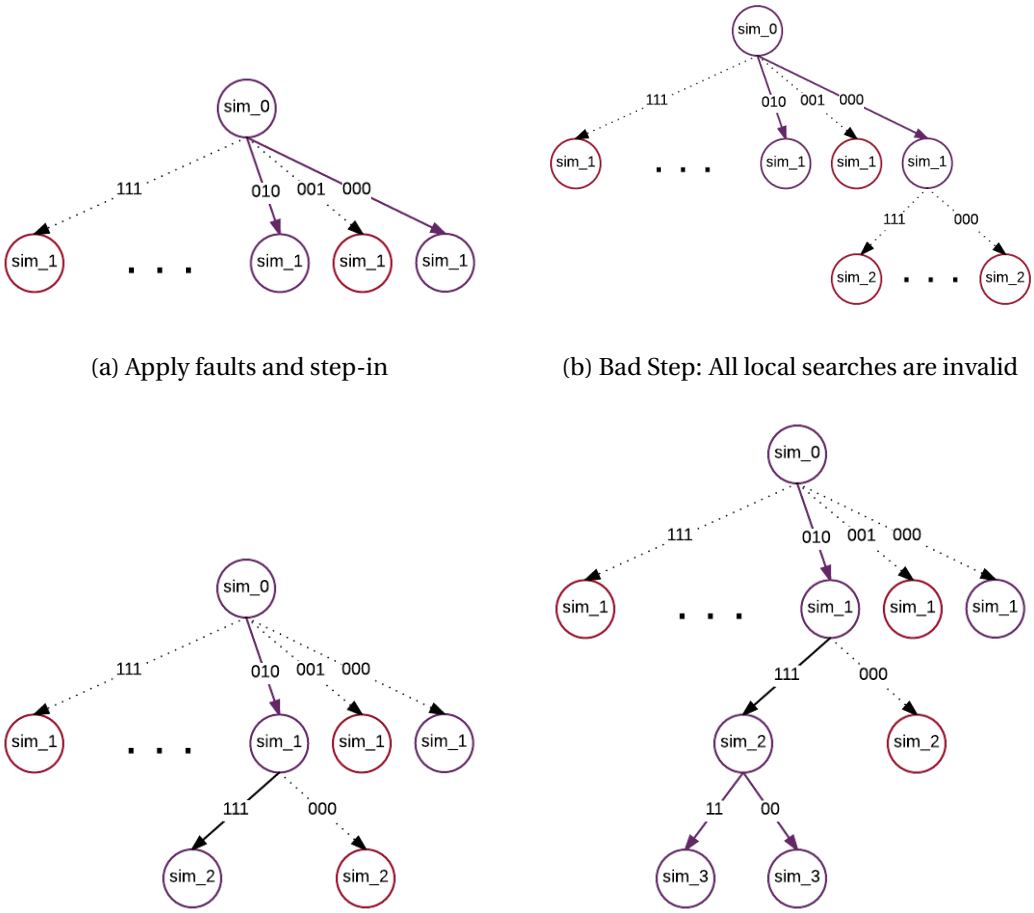
3 Algorithm and Implementation

This section generalizes the functional satisfiability technique and describes its parallel implementation.

3.1 Functional Satisfiability: General

Given N mystery gates, we know that at any input vector, we'd need to insert 2^N faults and test the design (eg: for 8 mystery gates, the local search space is 256, as those many unique faults could be assigned to those gates combined). This incremental application of input vectors over the local search space, is termed *stepping-in* (Figure 3.1a).

A `Good Step` is one where the golden output matches the reference output, and where the clauses for each mystery gate are non-empty. An empty clause indicates that no gate in the library is able to output the required execution trace for the simulated gate input. A `Bad Step` is an exploration where the reference output is invalid against the golden, or at least one of the mystery gate clauses is empty (Figure 3.1b). Whenever, an unit clause is obtained for a mystery gate, we assign that gate to its final suspect and no longer introduce faults at



(a) Apply faults and step-in

(b) Bad Step: All local searches are invalid

(c) Continue exploration along Good Steps

(d) Dynamically shrinking local search space

Figure 3.1: Exploratory Functional SAT for N=3

that gate. Therefore, the local search space dynamically shrinks as the exploration continues (Figure 3.1d), and its size can be redefined as $2^{N'}$, where N' is the number of non-unit clause mystery gates. Just as with the simple case, from the previous section, once all mystery gates have unit clauses, the exploration is terminated and the algorithm switches to verification of that solution (simple assign the suspect to the gate, and apply input vectors). The pseudo-code algorithm is described in Figure 3.2.

```
mgs_sat(test_index, sim){

    if(test_index == last && mystery_gates == convereged){
        #pragma omp critical
        record_result()

    } else {
        my_inputVector = inputvector[test_index]
        my_outputVector = outputvector[test_index]

        if(mystery_gates == convereged){
            // Exploration Phase

            foreach(fault){
                my_sim = sim    //clone sim object

                my_sim.tieValue(fault, mystery_gates)
                my_sim.applyVector(my_inputVector)
                my_sim.step_in()

                // golden vs reference
                if(my_outputVector == my_sim.Output()){
                    my_sim.reduce_clause(mystery_gates)
                    #pragma omp task
                    mgs_sat( (test_index+1), my_sim)
                }
            }

        } else{
            // Verification Phase
            my_sim = sim
            my_sim.applyVector(my_inputVector)
            my_sim.step_in()
            mgs_sat( (test_index+1), my_sim)
        }
    }

}
```

Figure 3.2: pseudo-code showing exploratory decomposition and parallelization

3.2 PARALLELIZATION

The algorithm is successfully parallelized using OpenMP. The recursive search of the local search space along optimal steps, intuitively represents *exploratory decomposition*. Discovery of further search space is non-deterministic and is only found during runtime. These explorations for subsequent step-ins are assigned as OpenMP Tasks upon the recursive function call.

The algorithm is implemented in way that each task (a node on the step-in exploration graph) performs its local search, and upon encountering a `Good Step`, it creates and adds an OpenMP Task for the next step-in to the task pool. Any thread that is free, picks up work from the pool, and executes the node operations. If all the local searches were bad, then the thread becomes free. Note that only single thread works upon entering the verification phase for some branch of converged solution, because this execution is inherently sequential.

To enforce maximum parallelization and to always keep the threads busy, a dynamic programming approach is employed. For the step-in of each fault at the local search, the simulator object[1] is cloned, and processed. This would be the object that would be passed to subsequent tasks for a Good Step. This ensures that each task has no dependency on each other (aside from the sequential discovery), and does not need to roll-back or maintain a history of execution. Therefore, the parallel implementation does not require any barriers or synchronization (except while recording the final result into a common data structure). The state of the simulator object grows dynamically, and at final converges, each object would have only executed each input vector only once.

4 Evaluation

In this section, we analyze the effects of parallelization and explore practical viability of the algorithm by emphasizing scalability. The algorithm is benchmarked on circuits from the ISCAS'89 set, with the following constraints:

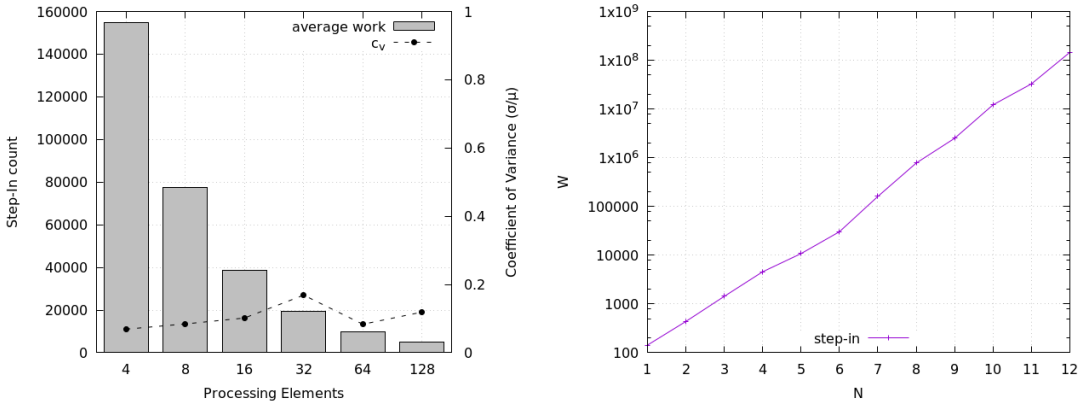
- Simulation traces are random, with length, $K = 128$
- Random obfuscation of gates of with, $input \geqslant 2$

4.1 Load Balancing

The greatest concern with exploratory decomposition is the balance of work among the processing elements (threads). Since, the discovery of the search space is non-deterministic, not all threads can begin or terminate work at the same time. Nonetheless, over the whole execution we rely on OpenMP's task scheduler to balance the work load between the threads,

[1] The simulator for this project was built upon the Event-Wheel simulator written by Dr. Michael S. Hsiao, with permission. The original simulator was completely overhauled to the C++11 standard and extended to include various algorithms described in this project, implemented in an OOP fashion.

with the dynamically maintained task pool. To illustrate this circuit c432 with 8 mystery gates is studied. Figure 4.1a shows that the average work (step-in count) distributes quite well among the threads, with increasing number of threads. Moreover, the coefficient of variance (c_v) of the work load being around 10% is satisfactory.



(a) Exploratory decomposition load, N=8 (b) Work(W, step-in count) vs Problem Size(N)

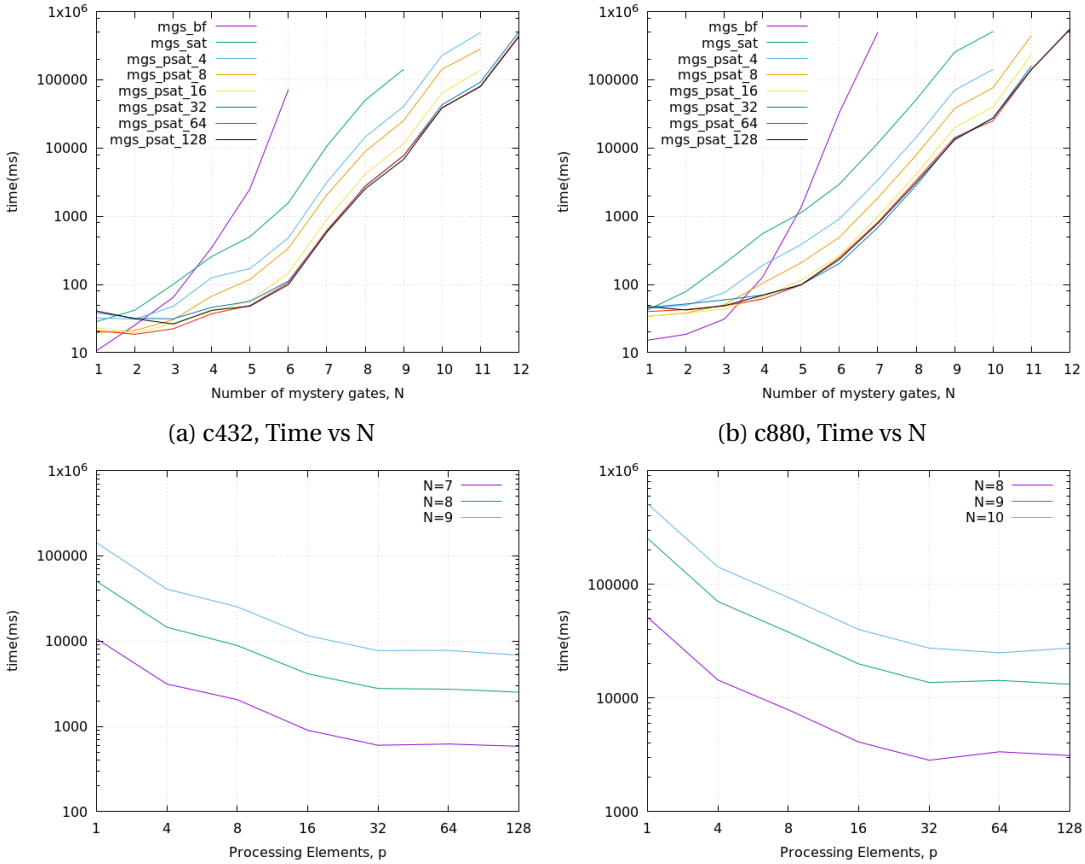
Figure 4.1: Load Balancing and Work Done, in c432

Figure 4.1b shows that with increasing problem size (number of mystery gates), the search space increases drastically, and solving for maximal SAT requires exponentially more step-in counts. Therefore considering that the problem size can only be controlled by the integer granularity of the number of mystery gates, the work to be done does not scale well for direct practical implementations (where the whole circuit is solved at once for all mystery gates).

4.2 Speedup and Efficiency

The execution speed of the base serial functional SAT implementation over the brute-force one is exponential. For example, for $N = 8$ in c432, `mgs_bf` takes 1 hour, while `mgs_sat` takes 1 minute. Parallelization affords further speedup with satisfactory efficiency with around 32 processing elements itself. Nonetheless the efficiency quickly tapers off beyond $p = 32$, and further speedup obtained is minimal. The algorithms are compared for execution time in Figure 4.2 for c432 (204 gates) and c880 (470 gates). The speedup and efficiency is displayed in Figure 4.3 and Figure 4.4. The trend is similar and indicates poor *strong scaling*. Note, that a timeout is maintained at 10 minutes, which also indicates the maximum problem size that can be solved in the given time constraint with increasing number of processing elements. The keys in the plots represent,

`mgs_bf` : Brute-Force (Figure 4.2a and Figure 4.2b)
`mgs_sat` : Serial Functional SAT
`mgs_psat_#p` : Parallel Functional SAT with p processing elements



(a) c432, Time vs N

(b) c880, Time vs N

(c) c432, Time vs p

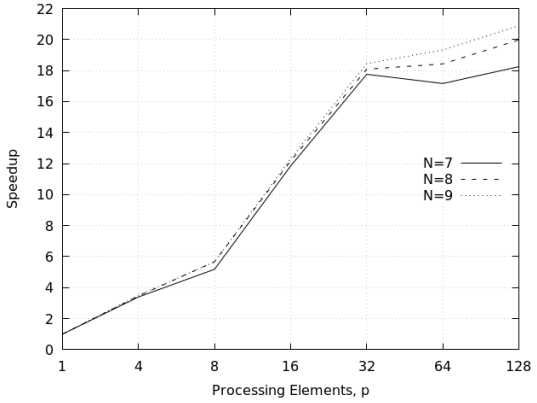
(d) c880, Time vs p

Figure 4.2: Execution Time

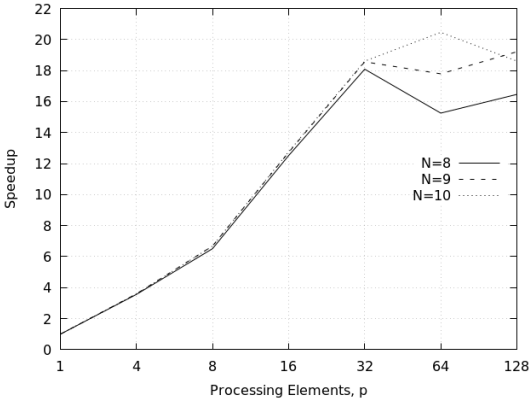
4.3 Scalability

The before-mentioned experimental speedup and efficiency graphs, clearly indicated poor *Strong scaling*, because for the same problem size, the execution time does not scale linearly with the number of processing elements (especially beyond $p = 32$).

Moreover, `mgs_psat` also shows poor *Weak Scaling*. We cannot attempt to solve large problem sizes (N), by increasing the number of processing elements (p). We see that the problem size does not grow proportionally with the number of processing elements. For this

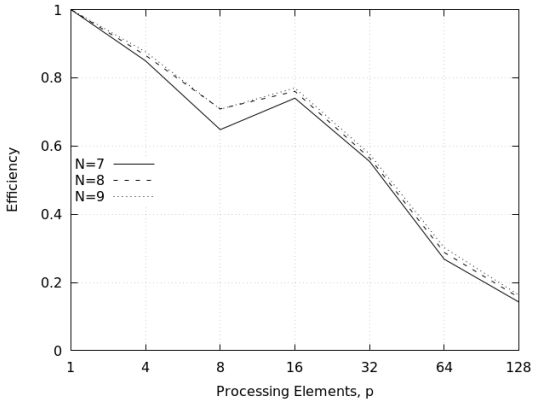


(a) c432

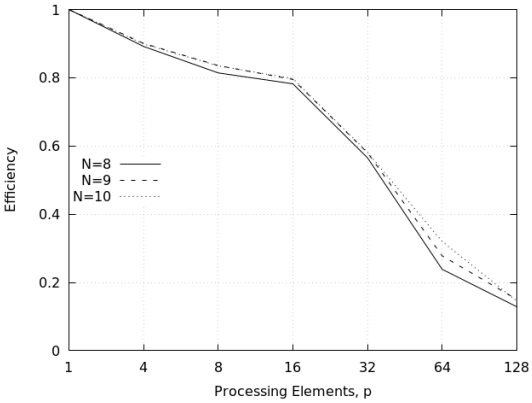


(b) c880

Figure 4.3: Speedup



(a) c432



(b) c880

Figure 4.4: Efficiency

analysis, we consider a constant upper-bound on the runtime (say, a timeout of 10 minutes), and tabulate (Table 4.1) the maximum problem size that can be solved within that constrain to experimentally demonstrate *Weak Scaling*.

5 CONCLUSION

Functional Satisfiability is certainly superior to the naive Brute-Force approach with exponential increase in base algorithmic execution time for maximal solutions. The characteristic local search of the algorithm was parallelized using exploratory decomposition over OpenMP

p	N
1	9
4	11
8	11
16	11
32	12
64	12
128	12
256	12

(a) T=10min,Fixed p, Maximum N

N	p
1	1
2	1
3	1
4	1
5	1
6	1
7	2
8	2
9	2
10	2
11	4
12	32
13	>256

(b) T=10min, Fixed N, Minimum p

Table 4.1: Weak Scaling Analysis of mgs_psat on c432

Tasks. Moreover, dynamically programming the exploration allowed each Task to operate without any need for synchronization, aside from the sequential discovery of tasks. Despite parallelization, the scalability of functional satisfiability is limited.

Good parallel speedup (18) is observed, but tapers off beyond $p = 32$, despite increasing problem size. Experimental scalability analysis showed that for $N \geqslant 12$, the number of processing elements required would not be practical. This could be attributed to the exponential growth of the search space with increasing problem size.

Therefore, future work could consider decomposing the problem into smaller chunks of the circuit, enclosing fewer mystery gates, and solving locally. At the end, these sub-results could be combined to form a much smaller global search space. We can also see that functional satisfiability could be achieved sooner by having optimal input excitation that activates the various states of the mystery gates allowing classification between its gate possibilities. This would require testbench generation that would always allow the mystery gate paths to be dominant.

REFERENCES

[1] W. Li, Z. Wasson, and S. A. Seshia. *Reverse Engineering Circuits Using Behavioral Pattern Mining*. In Proc. of HOST 2012.

[2] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik. *Reverse engineering digital circuits using functional analysis*. In Proc. of the DATE 2013.