Parallel vs Serial Fault Simulator

Michalis Christou

May 18, 2020

Introduction

The purpose of this project is to write a fault simulator in C++ that has both parallel and serial modes of operation. The goal is to examine the speedup of the parallel implementation compared to the serial version as well as compare any potantial speeups that can occur with or without fault dropping in both cases.

Backround

Single Stuck at Fault Model

Faults in circuits can be modeled in many ways. The model that was used for the purposes of this project was the single stuck at fault model. In this case the circuit, which is a gate level netlist, has only one faulty line that is either stuck permanently at 1 or 0 and the fault can be at an input or output of a gate. The number of possible fault sites in a given netlist is $\#PI + \#GATES + \#(fanoutbranches)$ and subsequentlty since a fault site has 2 potential faults, the number of possible faults in a circuit is $2 \cdot \#(faultsites)$.

Fault Equivalence

Despite a circuit having a maximum number of possible stuck at faults, using some mechanisms we can choose a subset of these faults and if we are able to detect these it means that we can also detect the ones we skipped. For example, fault equivalence means that two faults f_1, f_2 are equivalent if all tests that detect f_1 also detect f_2 and vice versa. This means that if we are able to detect one of the two faults then we know doe sure that we can detect the other. Thus we can keep only one of these equivalent faults in our list. This is called fault collapsing. The results is a collapsed fault set that contains one fault from each equivalence subset.

Fault Dominance

The definition of fault dominance is as follows, if all tests of some fault f_1 detect another fault f_2 then f_2 is said to dominate f_1. The rule that we use is if fault f_2 dominates f_1 then f_2 is removed from the fault list.

Checkpoint Theorem

The checkpoint theorem is the result of Fault Equivalence and Fault Dominance. Primary inputs and fanout branches of a combinational circuit are called checkpoints. The theorem suggests that

a test set that detects all single stuck at faults on all checkpoints of a combinational circuit also detects all single stuck at faults in that circuit. The checkpoints in a circuit are considered to be primary inputs and fanout branches. This means that if we only consider these fault sites and detect all faults in them, we can also detect all possible stuck at faults in that circuit.

Fault Simulation

Fault simulation is done in order to detect possible faults in a Boolean circuit. The algorithm is fairly simple and intuitive. First a set of test vectors is obtained. Then using these test vectors the non-faulty circuit response is computed. Then for each stuck at fault that is considered, the fault is injected in the circuit, the fault output is computed and compared to the normal non-faulty output. If the outputs differ, then the fault can be detected. In this case a detected fault counter is incremented. If fault dropping is enabled, then the stuck at fault is deleted from the set. After this is repeated for all test vectors and all faults in the set the fault coverage is computed which is the number of detected faults divided by the total number of faults considered.

$$FC = \frac{\#fd}{\#tf}$$

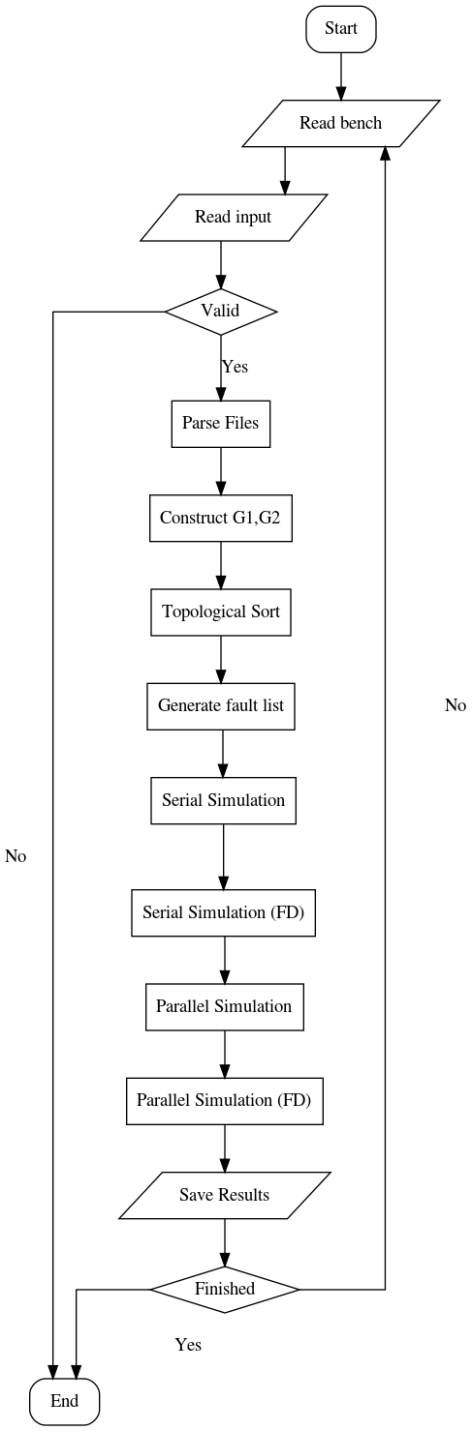
The pseudocode for the serial algorithm is depicted below.

Algorithm 1 Serial Fault Simulation

```
1: #fd = 0, #tf = |F|
2: for every test t ∈ T do
3:     Rt = true_value_simulation(t, C)
4:     for every fault f ∈ F do
5:         inject_fault(C, f)
6:         Rf = fault_simulation(C, f)
7:         if Rt ≠ Rf then
8:             #rf + +
9:             F = drop_fault(t, F)
10:        end if
11:    end for
12: end for
13: return FC = compute_fault_coverage(#fd, #tf)
```

Implementation

This section is dedicated to explain the details and inner workings of the specific C++ implementation. The code is designed to parse `.bench` files that represent gate level netlists as well as test vectors of different formats. Then the code generates all faults according to the Checkpoint Theorem. Depending on the user input the program either runs the serial algorithm or parallel with or without fault dropping. By utilizing bash scripts the simulator can be used via it's command line arguments. The following in how a collection of bash scripts and the simulator program work together to produce the results.



Netlist Representation

The netlist is internally represented as a DAG (Directed Acyclic Graph). Specifically, since the purpose of the program is to simulate faults, the graph model that was chosen was G_2. This model uses one graph node per circuit line and one edge per pair of gates/fanout stems with direct connection, in order to explicitly represent fanout stems. This is necessary in this case because faults could happen in any circuit line thus explicit representation of every line is needed. The data structure used in this case was a $1 - D$ array with linked lists, since generally speaking, Boolean circuits are sparse thus more memory is saved compared to using a $2 - D$ array.

Parallelizing Fault Simulation

When parallelizing algorithms a lot has to be considered. Data dependencies is one aspect that leads to race conditions which in turn, leads to unpredictable code behaviour. In this case, the most important part of parallelizing the algorithm is to ensure that no data dependencies or race conditions would occur. In this case this was achieved easily by splitting the work in equal chunks to N threads. Since each iteration is almost fully independent splitting the workflow to any number of threads was trivial once the serial algorithm was implemented. The workflow was split as follows: Each thread received a subset of the test vector set, then independently worked on identifying all faults that were previously generated with the checkpoint theorem and stored in a global variable. When using fault dropping threads that detect faults need to modify a global variable (that stores faults) in order to save time for other threads. To ensure that threads would not interfere with one another the global data structure was protected with a mutex since it was part of a critical region, but this was only a problem when using fault dropping. Without fault dropping each thread can work fully independent from any other one.

CLI Arguments

Then the simulator is used in the following fashion it just uses the benchmark file and corresponding input file given by the user, computes the fault coverage in serial mode and prints useful statistics in stdout including the fault coverage. This is the most basic usage.

```
./simulator -C <benchmark_path> -I <test_path> -S
```

The `-S` parameter can be combined with the `-T <[s],[ms],[us],[ns]>` argument in order to get the total runtime of the program as well as the netlist stats in stdout. The user can also choose the time unit. The next example prints netlist stats as well as the total runtime in seconds.

```
./simulator -C <benchmark_path> -I <test_path> -S -T s
```

Apart from computing the fault coverage the simulator can output the test vector that detected faults in a txt file via the `-H` argument. The usage in this case is as expected.

```
./simulator -C <benchmark_path> -I <test_path> -H
```

The next argument tells the simulator in which mode to run (serial or parallel) and to how many threads to allocate the work. If no number of theads is given then the simulator chooses the maximum number of concurrent threads supported by the CPU. In this instance the simulator will run in parallel on 8 threads.

```
./simulator -C <benchmark_path> -I <test_path> -P 8
```

Another useful argument is `-D` which enables fault dropping mode. This can dramatically decrease the simualtion time while still giving out correcr results. It's an optimization step that is useful for comparing test vector quality.

```
./simulator -C <benchmark_path> -I <test_path> -D
```

An argument that doesn't change the functionality of the simulator but comes in handy when working wit large netlists is the `-B` option. This option gives user feedback about the progress of the simulator, estimated execution time and how many test vectors per second are applied to the netlist. The argument is used like so:

```
./simulator -C <benchmark_path> -I <test_path> -B
```

For example, by running the simulator with the following arguments:

```
./simulator -C s5378.bench -I s5378.vec -B
```

We get the following output:

```
[100 P/s][total 0][0% done][100% in 74 min]
```

But when running the simulator in parallel:

```
./simulator -C s5378.bench -I s5378.vec -B -P 8
```

We get the following:

```
[500 P/s][total 0][0% done][100% in 15 min]
```

This is great for user feedback as well as a quick first look of how well the work is parallelized. I this case we can clearly see that when using the parallel mode, the simulator will finish approximately 5 times faster. Do note that all arguments can be combined. In the following example the simulator will output stats as well as total runtime in stdout when it's done, use fault dropping and run in parallel on 8 concurrent threads.

```
./simulator -C <benchmark_path> -I <test_path> -T s -S -D -P 8
```

Netlist Statistics

The table below shows some useful netlist statiastics for all benchmark circuits. The table includes circuit data from both ISCAS85 and ISCAS89 benchmarks.

Circuit	Nand	And	Nor	Or	Not	Buff	PI	PO	Paths	KB
s27	1	1	4	2	2	0	7	4	28	5
s386	0	83	0	35	41	0	13	13	207	77
s510	61	34	55	29	32	0	25	13	369	100
s641	4	90	0	13	272	0	54	42	1722	130
s838.1	57	105	70	56	158	0	66	33	1714	186
s298	9	31	19	16	44	0	17	20	231	61
s382	30	11	34	24	59	0	24	27	400	78
s400	36	11	34	25	56	0	24	27	448	82
s526	22	56	35	28	52	0	24	27	410	106
s713	28	94	0	17	254	0	54	42	21812	145
s344	18	44	30	9	59	0	24	17	344	65
s344.1	18	44	30	9	59	0	24	17	344	65
s420.1	29	49	34	28	78	0	34	17	474	91
s820	54	76	66	60	33	0	23	24	492	163
s953	114	49	112	36	84	0	45	52	1156	193
s208.1	15	21	16	14	38	0	18	9	142	43
s349	19	44	31	10	57	0	24	17	354	66
s444	58	13	34	14	62	0	24	27	535	90
s1238	125	134	57	112	80	0	32	32	3559	245
s1494	0	354	0	204	89	0	14	25	976	293
s5378	0	0	765	239	1775	15	214	228	13542	1064
s832	54	78	66	64	25	0	23	24	506	165
s1423	64	197	92	137	167	0	91	79	44726	289
s1196.1	119	118	50	101	141	0	32	32	3098	236
s1488	0	350	0	200	103	0	14	25	962	292
s9234	528	955	113	431	3570	0	247	250	244854	1823
s13207	849	1114	98	512	5378	0	700	790	1345369	2683
s15850	968	1619	151	710	6324	0	611	684	164738046	3176
s38417	2050	4154	2279	226	13470	0	1664	1742	1391579	7703
s38584	2126	5516	1185	2621	7805	0	1464	1730	1080723	7734
s35932	7020	4032	0	1152	3861	0	1763	2048	197141	7249
c17	6	0	0	0	0	0	5	2	11	3
c880	87	117	61	29	63	26	60	26	8642	174
c1355	416	56	0	2	40	32	41	32	4173216	267
c1908	377	63	1	0	277	187	33	25	729056	376
c2670	254	332	12	77	321	305	233	140	679954	560
c3540	298	495	68	92	490	246	50	22	28265874	689
c5315	454	718	27	214	581	419	178	123	1341305	1068
c6288	0	256	2128	0	32	0	32	32	1101055638	1218
c7552	1028	776	54	244	876	593	207	108	726494	1486

Fault Coverage

The results depicted below were obtained using the developed simulator. Note that the fault coverage is evaluated for each test set basis and not for each circuit. The tables below show the results for both ISCAS-89 and ISCAS-85 benchmark suits.

ISCAS-89

Circuit	Fault Coverage (%)	Test Vector
s27	87.5 %	s27.vec
s386	25.99 %	s386.vec
s510	88.96 %	s510.vec
s641	62.21 %	s641.vec
s838.1	32.52 %	s838.vec
s298	74.02 %	s298.vec
s382	72.77 %	s382.vec
s400	66.6 %	s400.vec
s526	57.81 %	s526n.vec
s526	64.11 %	s526.vec
s713	61.72 %	s713.vec
s344	70.18 %	s344.vec
s344.1	70.18 %	s344.vec
s420.1	41.67 %	s420.vec
s820	42.18 %	s820.vec
s953	57.8 %	s953.vec
s208.1	58.33 %	s208.vec
s349	72.35 %	s349.vec
s444	68.44 %	s444.vec
s1238	57.88 %	s1238.vec
s1494	64.23 %	s1494.vec
s5378	74.66 %	s5378.vec
s832	43.3 %	s832.vec
s1423	65.6 %	s1423.vec
s1196.1	63.87 %	s1196.vec
s1488	66.35 %	s1488.vec
s13207	66.94%	s13207.vec
s15850	67.29%	s15850.vec
s38417	72.84%	s38417.vec
s38584	75.31%	s38584.vec
s35932	71.57%	s35932.vec

Circuit	Fault Coverage (%)	Test Vector
s344	100.0 %	s344.tests.comp.10
s344.1	100.0 %	s344.tests.comp.10
s386	100.0 %	s386.tests.comp.10
s510	100.0 %	s510.tests.comp.10
s526	99.85 %	s526.tests.comp.10
s641	100.0 %	s641.tests.comp.10
s820	100.0 %	s820.tests.comp.10
s953	80.47 %	s953.tests.comp.10
s1196	99.78 %	s1196.tests.comp.10
s1423	98.76 %	s1423.tests.comp.10
s1488	100.0 %	s1488.tests.comp.10

Circuit	Fault Coverage (%)	Test Vector
s344.1	100.0 %	s344.benchtest.in
s344	100.0 %	s344.benchtest.in
s382	100.0 %	s382.benchtest.in
s510	100.0 %	s510.benchtest.in
s526	99.85 %	s526.benchtest.in
s641	100.0 %	s641.benchtest.in
s820	100.0 %	s820.benchtest.in
s953	74.19 %	s953.benchtest.in
s1196	99.78 %	s1196.benchtest.in
s1423	98.89 %	s1423.benchtest.in
s1488	100.0 %	s1488.benchtest.in

ISCAS-85

Circuit	Fault Coverage (%)	Test Vector
c17	100.0 %	c17.txt
c880	100.0 %	c880.txt
c1355	97.53 %	c1355.txt
c1908	99.71 %	c1908.txt
c2670	96.03 %	c2670.txt
c3540	96.01 %	c3540.txt
c6288	99.34 %	c6288.txt
c7552	98.38 %	c7552.txt

Circuit	Fault Coverage (%)	Test Vector
c17	100.0 %	c17.vec
c880	100.0 %	c880.vec
c1355	99.51 %	c1355.vec
c1908	72.27 %	c1908.vec
c2670	95.97 %	c2670.vec
c3540	81.58 %	c3540.vec
c5315	99.0 %	c5315.vec
c6288	99.34 %	c6288.vec
c7552	98.33 %	c7552.vec

Random Test Vectors

The following results were obtained with the randomly generated test vectors that the simulator can produce. The fault coverage is depicted for each circuit is depicted below.

Circuit	Fault Coverage (%)	Test Vector
s27	81.25 %	s27.rand.txt
s386	39.87 %	s386.rand.txt
s510	83.28 %	s510.rand.txt
s641	65.31 %	s641.rand.txt
s838.1	33.33 %	s838.1.rand.txt
s298	73.18 %	s298.rand.txt
s382	76.56 %	s382.rand.txt
s400	74.79 %	s400.rand.txt
s526	62.76 %	s526.rand.txt
s713	62.81 %	s713.rand.txt
s344	70.78 %	s344.rand.txt
s344.1	70.78 %	s344.1.rand.txt
s420.1	47.5 %	s420.1.rand.txt
s820	58.19 %	s820.rand.txt
s953	58.69 %	s953.rand.txt
s208.1	53.95 %	s208.1.rand.txt
s349	70.29 %	s349.rand.txt
s444	69.01 %	s444.rand.txt
s1238	59.38 %	s1238.rand.txt
s1494	73.85 %	s1494.rand.txt
s5378	77.78 %	s5378.rand.txt
s832	57.98 %	s832.rand.txt
s1423	66.91 %	s1423.rand.txt
s1196.1	63.64 %	s1196.1.rand.txt
s1488	74.55 %	s1488.rand.txt
s9234	50.04 %	s9234.rand.txt
s13207	67.69%	s13207.rand.txt
s15850	68.87%	s15850.rand.txt
s38417	73.31%	s38417.rand.txt
s38584	74.74%	s38584.rand.txt
s35932	74.23%	s35932.rand.txt
c17	90.91 %	c17.rand.txt
c880	89.54 %	c880.rand.txt
c1355	87.82 %	c1355.rand.txt
c1908	76.51 %	c1908.rand.txt
c2670	79.93 %	c2670.rand.txt
c3540	86.43 %	c3540.rand.txt
c6288	91.37 %	c6288.rand.txt
c7552	83.22 %	c7552.rand.txt

Serial vs Parallel

The program was tested on an Intel i3-4170 with a frequency of 3.7GHz. The specific model is a dual core processor with hyperthreading yielding 4 logical threads that the simulator can take advantage of.The following simulations were run with Test Suite 1 for ISCAS-89 and Test Suite 2 for ISCAS-85. Another factor to consider is that the simulations were run without utilizing fault dropping.

Circuit	Serial (s)	Parallel (s)	Speed-Up
s27	0.022888	0.016533	138 %
s386	19.188	9.60941	199 %
s510	28.9047	14.3244	201 %
s641	12.7631	6.08636	209 %
s838.1	119.137	59.9089	198 %
s298	4.39661	2.41235	182 %
s400	8.23653	3.9386	209 %
s382	7.7328	3.86674	199 %
s526	30.7068	14.9828	204 %
s713	16.8729	8.52973	197 %
s344	2.49324	1.28374	194 %
s344.1	2.49429	1.29227	193 %
s420.1	16.3338	8.45377	193 %
s820	140.489	67.7029	207 %
s953	144.769	69.1739	209 %
s208.1	2.35459	1.25504	187 %
s349	2.57886	1.34936	191 %
s444	9.96129	4.80381	207 %
s1238	385.448	186.436	206 %
s1494	439.093	211.106	207 %
s5378	4649.32	2262.31	205 %
s832	147.638	72.6281	203 %
s1423	76.8411	36.5896	210 %
s1196.1	317.072	152.82	207 %
s1488	432.646	209.133	206 %
c17	0.00915	0.004667	196 %
c880	194.654	91.2775	213 %
c1355	732.062	359.498	203 %
c1908	921.572	455.084	202 %
c2670	1471.39	728.113	202 %
c3540	8001.94	3931.87	203 %
c5315	2197.81	1060.8	207 %
c6288	1004.8	488.657	205 %
c7552	8340.21	3794.6	219 %

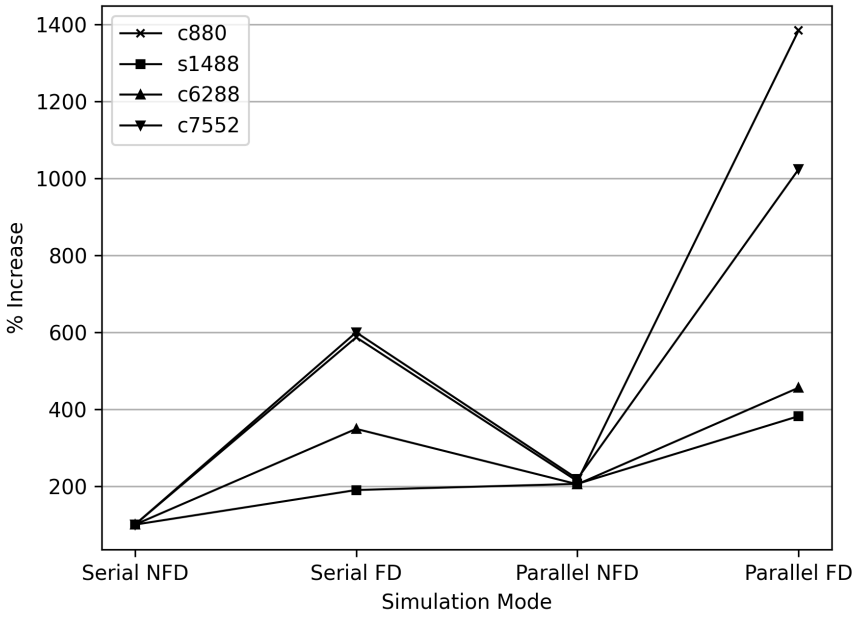
Fault Dropping

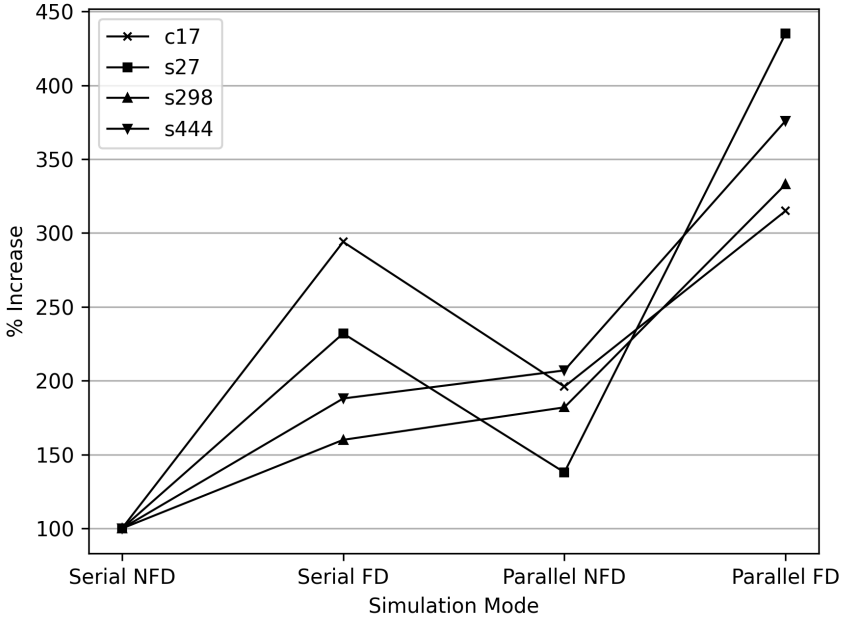
The results below were obtained by running the simulator on a single thread with and without fault dropping. Just like in the previous comparison, the simulator was run with Test Suite 1 for ISCAS-89 and Test Suite 2 for ISCAS-85.

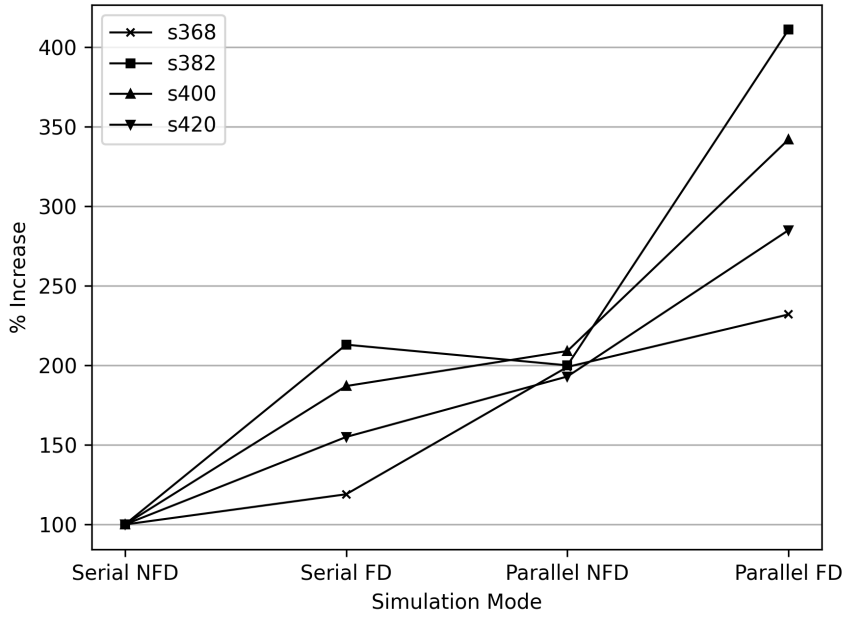
Circuit	Not FD (s)	FD (s)	Speedup
s27	0.022888	0.009834	232 %
s386	19.188	16.1131	119 %
s510	28.9047	9.71582	297 %
s641	12.7631	6.64599	192 %
s838.1	119.137	83.541	142 %
s298	4.39661	2.74366	160 %
s382	7.7328	3.6311	212 %
s400	8.23653	4.38963	187 %
s526	30.7068	16.3312	188 %
s713	16.8729	9.17201	183 %
s344	2.49324	1.29771	192 %
s344.1	2.49429	1.29767	192 %
s420.1	16.3338	10.5376	155 %
s820	140.489	104.111	134 %
s953	144.769	80.4509	179 %
s208.1	2.35459	1.27656	184 %
s349	2.57886	1.55484	165 %
s444	9.96129	5.28987	188 %
s1238	385.448	206.987	186 %
s1494	439.093	245.723	178 %
s5378	4649.32	1612.79	288 %
s832	147.638	106.458	138 %
s1423	76.8411	40.4679	189 %
s1196.1	317.072	166.549	190 %
s1488	432.646	227.759	189 %
c17	0.00915	0.003105	294 %
c880	194.654	33.4583	581 %
c1355	732.062	140.185	522 %
c1908	921.572	339.089	271 %
c2670	1471.39	238.299	617 %
c3540	8001.94	2031.53	393 %
c5315	2197.81	547.946	401 %
c6288	1004.8	287.604	349 %
c7552	8340.21	1386.35	601 %

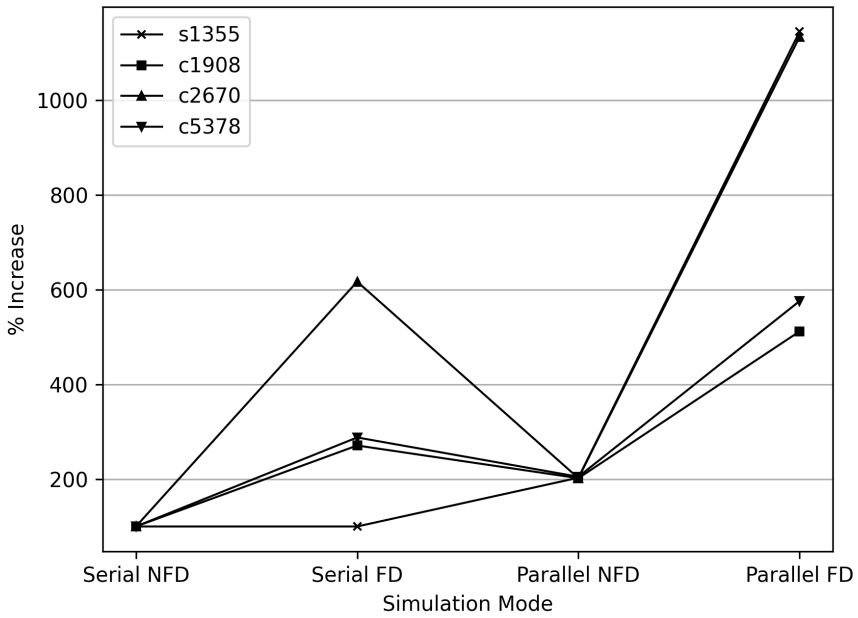
Performance Comparison

The figures below compare the performance increase for a subset of the ISCAS-89 and ISCAS-85 benchmark suits for serial, parallel, with and without dropping.






s368
s382
s400
s420
400
350
300
250
200
150
100
% Increase
Serial NFD
Serial FD
Parallel NFD
Parallel FD
Simulation Mode


s1355
c1908
c2670
c5378
1000
800
600
400
200
% Increase
Serial NFD
Serial FD
Parallel NFD
Parallel FD
Simulation Mode

Conclusions

As we can see, the program achieves roughly 200 % speedup when running in 4 parallel threads compared to running on a single one. Despite not reaching the theoritical maximum of 400 % a noticable performance increase was observed. Many factors could be limiting the performance increase. One of them is the fact that when a thread detects a single stuck at fault, it accesses a global variable where all detected faults are stored. If the data structure is in use by another thread, then the thread blocks. This is a limiting factor of the current implementation. When looking at the comparison with and without fault dropping a different trend is observed. Specifically, some circuit/test set pairs have a lot more speedup than others. This can be explained by the fact that the potential increase in speed depends solely on how good is the test vector set at detecting stuck at faults. If set of test vectors contains bitstrings that detect only s small number or no faults at all, then more no faults are dropped from the list thus the runtime of the program approaches the one without fault dropping.