High-Performance Parallel Fault Simulation for Multi-Core Systems

Masoomeh Karami*, Mohammad-hashem Haghbayan*, Masoumeh Ebrahimi†, Hamid Nejatollahi‡,
Hannu Tenhunen†, and Juha Plosila*
*Department of Future Technologies, University of Turku, Turku, Finland.
†Department of Electronics and Embedded Systems, Royal Institute of Technology (KTH), Kista, Sweden.
‡University of California, Irvine, CA USA.
Emails: *{mkaram, mohhag, juplos}@utu.fi †{mebr,hannu}@kth.se ‡ hnejatol@uci.edu

***Abstract*—Fault simulation is a time-consuming process that requires customized methods and techniques to accelerate it. Multi-threading and Multi-core approaches are two promising techniques that can be exploited to accelerate the fault simulation process by using different parts of the hardware at the same time. However, an efficient parallelization is obtained only by the refinement of software with respect to the hardware platform. In this paper, a parallel multi-thread fault simulation technique is proposed to accelerate the simulation process on multi-core platforms. In this approach, the gate input values are independently assigned to each thread. Each input value carries the information of several parallel simulation processes. This provides a multi-thread parallel fault simulation environment. The experimental results show that the proposed technique can efficiently use the hardware platform. In a single-core platform, the proposed technique can reduce the time by 25% while in a dual-core increasing the thread approximately halves the execution time.**

***Index Terms*—Multi-core system, multi-threading, parallelization, fault simulation**

I. Introduction

Fault simulation (FS) plays an essential role in different fields, such as test pattern generation, built-in-self-test, controllability, and observability analysis [1], [2]. This process is a challenging and time-consuming task in the VLSI design. In FS, a circuit-under-test (CUT) is simulated for a given fault model and a set of test patterns. This process is often computationally intensive, particularly for modern systems that require a large number of test patterns.

One of the most popular fault models for VLSI circuits is the stuck-at fault model [2]. Several works have used parallel and concurrent FS approaches to minimize the FS execution time for the stuck-at fault model in single-core [2], [3] and multi-core processors [4], [5]. They also use different methods such as *mixed-level fault simulation*, *parallelization*, and *event-driven* to accelerate the FS process. *Mixed-level* FS is an approach where non-faulty parts of a circuit are simulated faster using a higher abstraction level, e.g., behavioral level [6]. As FS has parallelizable characteristics, *parallel processing* has been widely used for minimizing the execution time. One parallelization technique is to partition the circuit into mutually exclusive parts and simulate them in parallel [7], [8]. Other parallelization solutions are based on simulating the circuit in parallel for different sets of faults (data-parallel) or test patterns (pattern-parallel) [9], [10].

There are two methods for implementing gate-level fault simulation: time-driven FS and event-driven FS. In the time-driven FS, the simulator calculates and updates all the gates' input/output variables in each small time epoch. In the event-driven FS, the gates' input/output variables are calculated based on the new occurred event, e.g., a change in the wire value. The event-driven method is more efficient compared to the time-driven method by being faster, using less memory, and being more flexible [11], [12].

In this paper, a parallel event-driven FS technique is proposed for multi-core systems to minimize the FS time. We describe the entire process from fault injection to scheduling tasks for parallel computing on different cores. The proposed technique is based on parallel processing of the occurred events in each level of the netlist.

II. Parallel event-driven fault simulation

In this section, we investigate two separate techniques to reduce the fault-simulation (FS) time, i.e., parallel FS [4], [5] and event-driven FS [13], [14], [11]. The considered fault model in this paper is stuck-at-0 (S@0) or stuck-at-1 (S@1) [2]. In a simple FS technique, after generating faults and eliminating the overlapping faults, i.e., fault collapsing [2], [15], faults are injected into the design. Then, the output of the design is compared with the expected value, i.e., golden output. If the output is the same as the golden output, it means the fault is not propagated to the output, and thus it is masked. Otherwise, it is said that the injected fault is observed and detected. For example, Figure 1 shows a 1-bit multiplexer where the output should be 0 when the signals $\{A$, B, and $SELECT\}$ are $\{000\}$. However, the S@1 fault on $wire\_3$ results in OUT=1, which is different from the golden result, and thus a fault is detected.

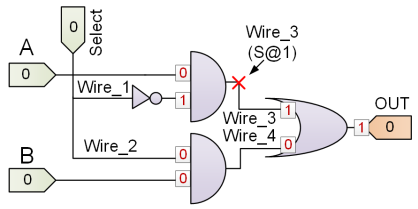


Fig. 1: An example of stuck-at FS in a 1-bit multiplier

A. Parallel fault simulation

One of the methods to accelerate the FS process is parallel FS [16], which is also utilized in this paper. In parallel FS, each bit in the input variables of a gate represents the propagation of an injected fault to the output gate. Then the output gate is evaluated by using logical operations in programming. Figure 2 depicts a parallel FS for the 1-bit multiplexer of Figure 1. Four fault propagation scenarios are simulated in parallel in Figure 2. The first scenario investigates the propagation of the S@1 fault on $Wire\_1$ (i.e., $wire\_1$ (S@1)). The output of this scenario equals to zero, which means the fault is masked. The other scenarios show that the golden result and the propagation of fault happened on $wire\_2$ (S@0) and $wire\_3$ (S@1). In this example, only the effect of $wire\_3$ (S@1) can be observed and detected on the output.

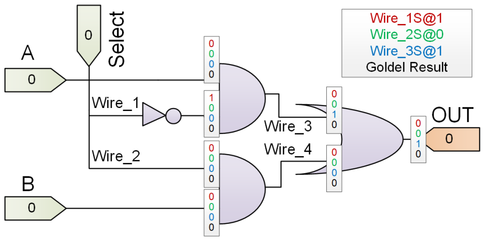


Fig. 2: An example of parallel FS in a 1-bit multiplier

B. Event-driven fault simulation

In an event-driven FS, the FS applies only to the gates on which an event has occurred instead of simulating all the gates. Thus it reduces the simulation time. Algorithm 1 depicts a pseudo code for an event-driven FS. At first, a $test\_pattern$ is injected into the simulation (line 1). Because our simulation is event-driven, all occurred events are pushed into the $STACK$ (line 2) in order to pop them in a right order and compute and update the variables (line 4). If the primary outputs are changed after injecting a fault (line 5), the fault is detected (line 6), so the event is pushed into the $STACK$ (line 7). The $STACK$ consists of the detected $events$.

Algorithm 1 The algorithm for an event-driven FS

```
Inputs: $test\_patterns$;
Body:
1: $events \leftarrow$ Inject_a_new_test_pattern_or_fault($test\_patterns$);
2: $STACK$.push ($events$);
3: while $STACK$.size() do
4:     $newEvents \leftarrow$ Evaluate gate ($STACK$.pop());
5:     if (Fault is injected and an event occurred on the primary output) then
6:         Fault is detected;
7:         $STACK$.push(occurred event);
```

The main issue with the event-driven FS is the situation caused by Reconvergence Fanout (RF). If an RF exists in the design, the evaluation of a gate should be delayed until all inputs of a gate are evaluated. Otherwise, glitches may occur in internal wires or the primary output. The issue increases the execution time and adds extra processes to fault detection. To address this problem, levelized event-driven simulation could be a viable choice that restricts the event evaluation to each level. In a levelized event-driven FS, first, the design is levelized, where the gates in each level get their inputs from the previous level. Each newly occurred event is placed in the list of its corresponding level. At each level, the gates in the event list are sequentially evaluated. Algorithm 2 illustrates this process in pseudo code. The implementation of the levelized fault-injectable circuits is automatic. In Algorithm 2, an array of dynamic stacks is employed for events occurring at each level. This process is continued until all test patterns are injected.

Algorithm 2 The algorithm for levelized event-driven FS

```
Inputs: $test\_patterns$;
Body:
1: $events \leftarrow$ Inject_a_new_test_pattern_or_fault($test\_patterns$);
2: for (all $event$ in $evets$) do
3:     $STACK$[level($event$)].push(event);
4: for ($i$ = $min\_level$ to $max\_level$) do
5:     while $STACK[i]$.size () do
6:         $newEvent \leftarrow$ Evaluate gate ($STACK[i]$.pop ());
7:         if (Fault is injected and an event occurred on primary output) then
8:             Fault is detected;
9:         for (all $event$ in $newEvent$) do
10:            $STACK[level(event)]$.push($event$);
```

The levelized event-driven FS can be combined with parallel FS. In this case, each *event* stack can be used for all events occurred by parallel faults. A levelized event-driven parallel FS method forms the base of our FS algorithm.

III. The Proposed Parallel Fault Simulation for Multi-Core Systems

A. Parallel Processing

The aim of parallel FS in multi/many core systems is to utilize available cores in the system as much as possible, and thus achieving a high performance. The challenge is to find the best method for task scheduling among cores. The main strategy to apply the fault in the proposed FS is to apply one test pattern and inject all available faults, and then repeat the process for other test patterns. Experimentally, we noticed that using this technique results in better parallelism compared to other equivalent methods. Figure 3 shows an example of the proposed method for parallel FS on a multi-core system. As it is shown, there are three cores available in the system, and three input variables are considered for each gate where the input variables contain the data for each thread. It is worth mentioning that the number of variables are not necessarily the same as the number of cores. Variables are 32-bit integer, and thereby, 32 FS scenarios can be performed per iteration by each thread. We call performing FS on each thread a *task*. Therefore, each task is assigned to one thread, and the operating system (OS) can schedule the thread for each task based on the available cores on the system [17].

Data dependencies among concurrent tasks may cause negative effect on the overall performance. Because of this scheduling of the threads is an essential step that should be performed carefully. On the other hand, task scheduling highly depends on the technique selected to perform FS algorithm. In the next section, we explain the process of task scheduling in the proposed algorithm.

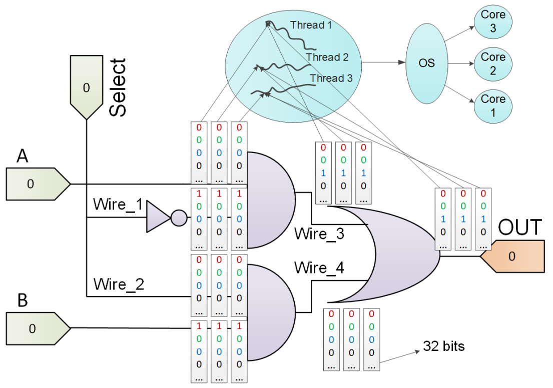


Fig. 3: The overall process of the proposed method

B. Task scheduling for parallel computing

One of the important factors in parallel computing is task scheduling. Task scheduling should balance the utilization of all cores in the system to reach the maximum concurrency. In the proposed task scheduling algorithm, first, the golden result for a test pattern is generated for all tasks. Then, a group of faults is applied to each task for FS. The fault groups should be independent of each other for concurrent tasks. After executing all threads, undetected faults will be accumulated. This process is continued until all faults are injected. Finally, the accumulated undetected faults from all threads will be collected to inject as the next test pattern. Algorithm 3 depicts in detail the proposed algorithm for multi-core FS and task scheduling. This algorithm models the proposed levelized multi-core event-driven FS that can be categorized into four main jobs as fault collapsing, fault grouping, fault injection, and fault simulation.

1) Fault injection process: As shown in Figure 4, the S@1 fault can be applied to a wire in a design by OR-ing the wire with logic 1. Similarly, the S@0 fault can be generated by AND-ing the wire with logic 0. The proposed fault-simulation algorithm sorts the events in a different way than that of Algorithm 2. In Algorithm 3, we disparate the memory of stimulated gates ($GATE\_STACK$) from the memory of stimulated faulty gates ($FAULT\_GATE\_STACK$). This is because of the fault injection method, in which the events related to the faulty gates in the next level are evaluated before those of the ordinary gates. By using this fault-injection method, some faults cannot be injected simultaneously for parallel FS to a thread. These faults (named dependent faults) should be categorized in different groups for allocation to different threads. The function called *Independent Fault Grouping* performs this categorization.

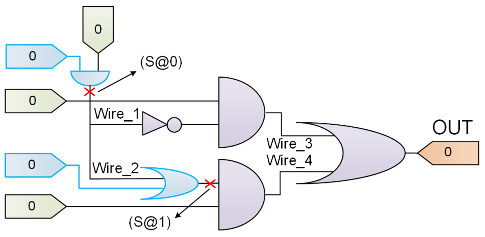


Fig. 4: An example of two dependent faults

Algorithm 3 The proposed algorithm for multi-core parallel FS

```
Inputs: $test\_patterns$;
Outputs: $FAULT\_GATE\_STACK$: stack of levelized fault events;
$GATE\_STACK$: stack of levelized events;
Body:
 1: Fault_Collapsing_Function();
 2: for (all $test\_pattern$ in $test\_patterns$) do
 3:     Independent_Fault_Grouping();
 4:     $*FAULT\_STACK \leftarrow$ Faultinjection($test\_pattern$);
 5:     Create_threads ($Number\_of\_Threads$);
 6:     for ($j$ = 1 to $Number\_of\_Threads$) do
 7:         Inject $FAULT\_STACK[j]$ to $thread[j]$ data structure;
 8:         //Performing event-driven parallel fault simulation
 9:         for ($i$ = $min\_level$ to $max\_level$) do
10:             while ($FAULT\_GATE\_STACK[i]$.size()) do
11:                 $newEvents \leftarrow$Evaluate gate($FAULT\_GATE\_STACK[i]$.pop());
12:                 if (a fault is injected and primary output is changed) then
13:                     Fault is detected;
14:             for (all $event$ in $newEvents$) do
15:                 $FAULT\_GATE\_STACK[level(event)]$.push ($fault\_event$);
16:                 $GATE\_STACK[level(event)]$.push ($event$);
17:             while ($GATE\_STACK[i]$.size ()) do
18:                 $newEvents \leftarrow$ Evaluate gate ($GATE\_STACK[i]$.pop ());
19:                 if (a fault is injected and primary output is changed) then
20:                     Fault is detected;
21:             for (all $event$ in newEvents) do
22:                 $FAULT\_GATE\_STACK[level(event)]$.push ($fault\_event$);
23:                 $GATE\_STACK[level(event)]$.push ($event$);
24:     Collect undetected faults to $*FAULT\_STACK$;
25:     Kill the generated threads();
```

2) Fault grouping: As was explained, for fault injection, ANDed with zero and ORed with one can be used to generate S@0 and S@1 faults, respectively. This method of fault injection limits the cascade of multiple faults for levelized event-driven FS. Figure 4 illustrates the process of fault injection with logic gates (called fault gates). In this figure, two faults are injected into the design,

In linear fault collapsing, the input and outputs for fanout have separate fault models that should be considered in FS individually [2], [18]. This situation may cause a cascade of faulty gates. In levelized event-driven FS, an occurred event in a level will be proceeded based on the level order. In each level, *FAULT_GATE_STACK* will be evaluated before *GATE_STACK*. If an event has occurred on a faulty gate, this will be stored on the next level *FAULT_GATE_STACK*. If two faulty gates are cascaded, the evaluation of the second fault will be missed in the next level. In fault grouping, dependent faults will be placed in different groups. In this case, dependent faults will be applied to different threads to run in parallel.

3) openmp.h for multi-core fault simulation: In the proposed method, shown in Algorithm 3, creating threads and determining the number of threads is essential. It is also necessary to distribute the collapsed faults among threads to achieve higher concurrency. Therefore, the optimal number of threads should be properly set, and critical sections should be determined. Based on the proposed algorithm, each gate should have some non-sharing variables for each thread. These copies of variables will be assigned to threads, and finally, the fault detection results will be gathered. We use openmp.h library of C++ to implement the algorithm. Algorithm 4 depicts the use of openmp.h functions for creating multi-thread fault propagation for one test pattern. The #pragma compiler generates the threads according to *thread_id*. The *fault_propagation* function starts the propagation of faults in each thread, and the detected faults will be then placed in its corresponding *thread_df*. The reduction parameter with plus argument means all number of detected faults for each thread should be added to the final value of *thread_df*. The *num_threads* determines the number of threads (according to *thread_id*) that should be created for the selected code.

Algorithm 4 The pseudo code for creating threads in concurrent fault propagation

```
Inputs: $thread\_id$;
Outputs: $detected\_faults$;
Body:
1: $fault\_number \leftarrow$ fault_injection($thread\_id$);
2: int $thread\_df$;
3: #pragma omp parallel reduction $(+ : thread\_df)$
4: for (num_threads ($thread\_id$)) do
5:     $thread\_df \leftarrow 0$;
6:     $thread\_df \leftarrow thread\_df$ + fault_propagation( $thread\_id$ );
7: $detected\_faults \leftarrow detected\_faults + thread\_df$;
```

The undetected faults should be collected and redistributed to threads for the simulation in the next round. However, collecting all undetected faults may cause conflict or race between threads for accessing the memory. To eliminate the race situation, the critical section definition is proposed, which is described in Algorithm 5. *faultStack_NDF* is the memory for collecting undetected faults. The critical parameter restricts the threads to perform the selected code concurrently. If task scheduling between threads is balanced, this critical section would not have a considerable impact on the performance.

Algorithm 5 The pseudo code for collecting undetected faults

```
Inputs: $FAULT\_STACK$;
Outputs: $faultStack\_NDF$;
Body:
1: #pragma omp parallel
2: while ($FAULT\_STACK[thread\_id]$.size()) do
3:     if ($FAULT\_STACK[thread\_id]$.top() is not detected) then
4:         $faultStack\_NDF[thread\_id]$.push(
           $FAULT\_STACK[thread\_id]$.top());
5:     $FAULT\_STACK[thread\_id]$.pop();
```

IV. Experimental Results

To evaluate the proposed FS method, several benchmarks are examined. We utilized *Netlist-Generator* [2] for evaluating the FS on the RTL-level HDL codes. Using this tool, the design is first modeled in the RTL format and then synthesized. We used open source version of Microsoft Visual Studio environment to invoke the openmp.h library. In addition, a tool is developed for converting the ISCAS benchmark circuits to the appropriate data structure for FS. To compare the performance of the proposed method with other existing methods, we have implemented the source code of the *hope* [13] fault simulator in the Visual Studio environment using dynamic features such as dynamic stack and link-list. The result obtained by this method is the same as the result obtained by the proposed method for one thread generation.

A. CPU Time of Single-core and Dual-core Systems

The executions were performed on a dual-core CPU with 2.2GHz clock cycles and 4GB RAM. To compare the results with a single core processor, a single core with 2.3 GHz CPU and 512 MB RAM is used. Table I (a) and (b) show the CPU time of FS with 20 random test patterns on the dual core and single core systems, respectively. Some general circuits, e.g., 8-bit and 16-bit multiplier, and ISCAS benchmarks are used as benchmarks. The average percent of fault coverage (FC) is 66.67%. As can be seen in both tables, in most cases, the CPU time decreases as the number of threads increases. This is because of the flexibility of the proposed method in utilizing the available cores in the system.

(a) Dual-core system

Benchmark	#Gates	Thread Number										
Name		1	2	3	4	6	8	10	20	30	40	FC%
8-bit multiplier	175	0.58	0.52	0.49	0.53	0.56	0.53	0.57	0.7	0.78	0.9	93.1
16-bit multiplier	787	5.67	3.6	3.32	3.21	3.29	3.35	3.54	3.91	4.47	4.88	93.5
C17	6	0.03	0.04	0.06	0.06	0.04	0.06	0.05	0.04	0.09	0.1	93.7
C432	160	0.59	0.63	0.58	0.61	0.64	0.65	0.69	0.83	0.98	1.1	56
C499	202	1.65	1.51	1.42	1.41	1.46	1.50	1.55	1.78	1.91	2.09	80.5
C880	383	2.53	2.33	2.17	2.27	2.28	2.37	2.37	2.82	3.13	3.34	63
C1355	546	5.09	4.26	3.76	3.90	3.95	3.96	4	4.55	4.96	5.35	75
C1908	880	6.22	5.05	4.57	4.52	4.62	4.72	4.78	5.30	6.13	6.64	60.8
C2670	893	40.08	31.3	28.3	28.7	28.8	29	29	29.6	31.4	33.2	62.3
C3540	1669	13.3	9.94	9.06	9.03	9.16	9.25	9.37	10	10.9	11.5	53.7
C5315	2307	89.4	58.3	52	52.6	52.7	52.7	57.5	53.6	55.3	57.1	65.9
C6288	2416	40.5	19	15.5	15.4	15.4	15.2	15.4	16.4	17.6	18.9	94.9
C7552	3512	134	77.7	67.3	66.9	66.8	66.8	66	67	69	71	69
S838_ex	446	4.01	3.74	3.42	3.49	3.53	3.58	3.62	4.29	4.52	4.85	33.4
S9234_ex	5597	184	166	165	165	165	164	163	165	168	170	38.6
S15850_ex	9772	1392	701	617	615	615	621	659	584	601	605	50.7

(b) Single-core system

Benchmark	#Gates	Thread Number										
Name		1	2	3	4	6	8	10	20	30	40	FC%
C432	160	1.43	1.51	1.51	1.51	1.59	1.68	1.78	2.1	2.53	2.86	56
C1908	880	14.1	13.3	13.1	13	13.2	13.3	13.7	15.2	16.6	18.1	60.8
C5315	2307	175	172	165	162	161	160	161	164	168	172	65.9

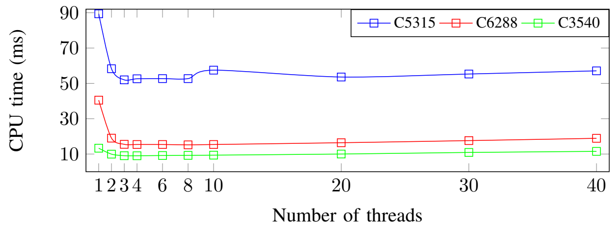
TABLE I: The result of CPU time for fault simulation on ISCAS benchmarks

In another experiment, we measured the core utilization while running a different number of threads for FS. Using three threads, the core utilization reaches the high usage of 97% while the usage is 50% and 89% when using one and two threads, respectively. This shows that doubling the number of threads does not necessarily double the utilization that is a common observation in multi-threading. The reason is due to synchronization, memory-access conflict, and other issues in a different layer of software. The amount of memory usage slightly increases when increasing the number of threads, i.e., 1.22 GB, 1.28 GB, and 1.30 GB for one, two, and three threads.

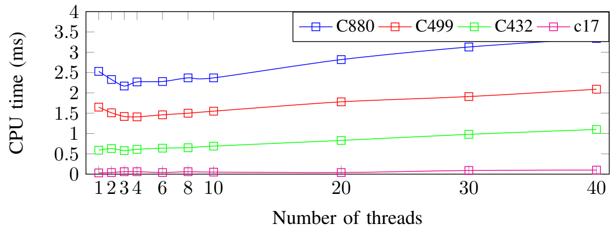
B. CPU Time versus the Number of Threads

Figure 5 (a) and (b) depict the CPU time with regard to the number of generated threads in a dual-core system when running three large-size and four small-size ISCAS benchmarks. This figure shows that the proper number of threads could offer an optimal CPU time. It can be seen that by increasing the number of threads, first the FS time sharply decreases, then it stabilizes, and finally, it starts rising again. It should be noted that, since there are two cores in the system, increasing the number of threads to more than two does not result in considerable execution time reduction.

Also, by having a large number of threads in the system, the overall communication time between threads and OS exceeds the time reduction archived by parallelism.



(a) Large-size ISCAS benchmark circuits



(b) Small-size ISCAS benchmark circuits

Fig. 5: CPU time versus the number of threads.

C. CPU Time with regard to the Number of Gates

As parallelism needs more data structures and communication overheads, the proposed method is not suitable for smaller designs but rather larger ones. Figure 6 shows the CPU FS time reduction with regard to the number of gates when applying the proposed method. It can be seen that the proposed method improves the FS time for larger designs more than the smaller ones.

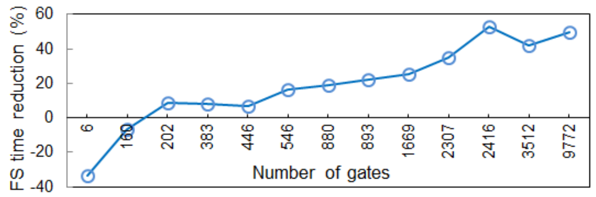


Fig. 6: FS time reduction versus the number of gates

D. CPU Time with regard to Multi-threading

The multi-thread FS may improve the FS CPU time for both dual-core and single-core systems. Figure 7 shows the FS time for C5315 on single and dual core systems. As can be seen in this figure, the CPU time can be reduced from 175ms to 160ms in a single-core system while this reduction is much larger in a dual-core system, i.e., from 89.4 ms to 52ms. The reason is due to the high level (OS level) of task parallelism but not task scheduling on individual cores.

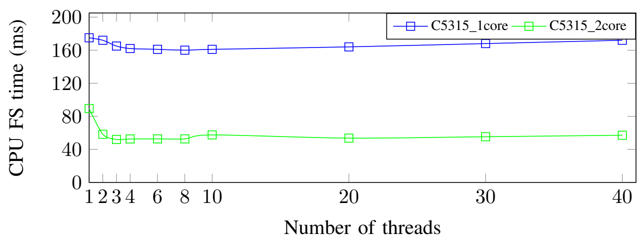


Fig. 7: CPU time of a single and dual core system for c5315

V. Conclusion

Fault simulation is a challenging algorithm w.r.t. its execution time and at the same time a critical step in the VLSI design. Reducing the fault simulation execution time plays an important role in developing better test patterns to debug the post silicon VLSI circuits. In this paper, we proposed a multi-thread parallel fault simulation method for multi/many core systems. One of the important steps in the proposed method is the task scheduling which defines how faults should be grouped, injected, and propagated. The proposed method is based on the levelized event-driven simulation. Results confirm the efficiency of the fault simulation method with regard to the number of cores, number of threads, and number of gates. As a future work, we plan to use this approach to analyze faults in applications such as neural networks running on a many/multi core systems.

References

[1] D. Lee and J. Na, "A novel simulation fault injection method for dependability analysis," in *IEEE Design and Test of Computers*, 2009, pp. 50–61.

[2] Z. Navabi, "Digital system test and testable design: Using hdl models and architectures," in *Springer Publisher*, 2010.

[3] N. Bombieri, F. Fummi, and V. Guarnieri, "Accelerating rtl fault simulation through rtl-to-tlm abstraction," in *ETS*, 2011, pp. 117–122.

[4] S. Hadjitheophanous, S. N. Neophytou, and M. K. Michael, "Scalable parallel fault simulation for shared-memory multiprocessor systems," in *VTS*, 2016, pp. 1–6.

[5] M. Gorev, R. Ubar, and S. Devadze, "Fault simulation with parallel exact critical path tracing in multiple core environment," in *DATE*, 2015, pp. 1180–1185.

[6] S. Mirkhani, M. Lavasani, and Z. Navabi, "Hierarchical fault simulation using behavioral and gate level hardware models," in *ATS*, 2002, pp. 374–379.

[7] A. Ehteram, H. Sabaghian-Bidgoli, H. Ghasvari, and S. Hessabi, "A simple and fast solution for fault simulation using approximate parallel critical path tracing," in *Canadian Journal of Electrical and Computer Engineering*, 2020, pp. 100–110.

[8] J. Kõusaar, R. Ubar, S. Kostin, S. Devadze, and J. Raik, "Parallel critical path tracing fault simulation in sequential circuits," in *MIXDES*, 2018, pp. 305–310.

[9] K. Gulati and S. Khatri, "Towards acceleration of fault simulation using graphics processing units," in *DAC*, 2008, pp. 822–827.

[10] R. Mueller-Thuns *et al.*, "VLSI logic and Fault Simulation on General-purpose Parallel Computers," in *IEEE Trans. on CAD of Integrated Circuits and Systems*, 1993.

[11] E. Gascard and Z. Simeu-Abazi, "Quantitative analysis of dynamic fault trees by means of monte carlo simulations: event-driven simulation approach," in *Reliability Engineering & System Safety*, 2018, pp. 487–504.

[12] J. A. Garrido, R. R. Carrillo, N. R. Luque, and E. Ros, "Event and time driven hybrid simulation of spiking neural networks," in *Advances in Computational Intelligence*, 2011, pp. 554–561.

[13] H. K. Lee and D. S. Ha, "Hope: an efficient parallel fault simulator for synchronous sequential circuits," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1996, pp. 1048–1058.

[14] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with gp-gpus," in *DAC*, 2009, pp. 557–562.

[15] M. Haghbayan, S. Teräväinen, A. Rahmani, P. Liljeberg, and H. Tenhunen, "Adaptive fault simulation on many-core microprocessor systems," in *DFTS*, 2015, pp. 151–154.

[16] J. Fan and Z. Zhang, "Speeding up fault simulation using parallel fault simulation," in *Procedia Engineering 15*, 2011, pp. 1817–1821.

[17] W. Stallings, "Operating systems," in *Prentice Hall*, 2001.

[18] M. H. Haghbayan, S. Karamati, F. Javaheru, and Z. Navabi, "Test pattern selection and compaction for sequential circuits in an hdl environment," in *ATS*, 2010, pp. 53–56.