An Efficient Parallel Fault Simulator for Functional Patterns on Multi-core Systems

Xiaoze Lin[1,2,3], Liyang Lai[4], Huawei Li[1,3], Biwei Xie[1,2], Xingquan Li[2]

[1]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2]Pengcheng Laboratory, Shenzhen, Guangdong, China
[3]University of Chinese Academy of Sciences, Beijing, China
[4]Electrical Engineering, Shantou University, Shantou, Guangdong, China

***Abstract*—Fault simulation targeting functional patterns emerges as an essential mechanism within functional safety, crucial for validating the effectiveness of safety mechanisms. The acceleration of fault simulation for functional patterns is imperative for boosting the efficiency and adaptability of functional safety verification, presenting a significant yet unresolved challenge. In the paper, we propose an efficient fault simulator for functional patterns, utilizing three techniques including fault filtering, fault grouping, and CPU-based parallelism. The integration of these three techniques, tailored to the characteristics of functional patterns, reduces the runtime of fault simulation from different perspectives. The experimental results show that on a 48-core system, an average 79x speedup can be achieved by our parallel fault simulator against a commercial tool.**

***Index Terms*—Fault simulation, Functional patterns, Fault filtering, Fault grouping, Parallel acceleration**

I. Introduction

Functional safety in chips ensures that electronic systems operate correctly and safely, even in the presence of hardware failures [1]. It is a critical aspect of design that addresses the reliability and safety requirements of systems, particularly in applications where failures could lead to hazardous situations. For instance, in the context of autonomous driving, a failure in the chip's functionality could result in the inability to execute critical safety maneuvers, highlighting the imperative need for functional safety measures to prevent such risky scenarios.

Fault simulation, crucial for identifying and mitigating faults within the chip, is an indispensable part of chip testing and verification. Within the context of functional safety, fault simulation is a vital tool for verifying the effectiveness of safety mechanisms in preventing, detecting, and mitigating risks [2].

Functional patterns, which are significantly different from scan patterns, are the patterns used in fault simulation within the functional safety domain. Scan patterns usually last for a maximum of several clock cycles, with each pattern being independent of the others. Whereas functional patterns, can cover tens of thousands to millions of clock cycles [3]. Within such extensive clock cycles, the logic states of the circuit's internal nodes hinge on the simulation results from preceding clock cycles, reflecting the circuit's functional behavior over time. Therefore, fault simulation for functional patterns requires continuously considering the fault effect over such a large number of clock cycles.

This work is supported by the Major Key Project of PCL under grant No. PCL2023A03, the National Natural Science Foundation of China (NSFC) under grant No. 92473203 and No. 92373206. (Corresponding author: Liyang Lai.)

Ensuring a chip's compliance with functional safety standards necessitates the iterative process of fault simulation, which demands that the simulation's runtime is at a manageable level. However, as the functional patterns can span millions of clock cycles, this essential process faces the significant challenge of extensive simulation time. For example, performing fault simulation on designs with hundreds of thousands of gates and million-cycle functional patterns can take several months. Based on this situation, the functional safety standard ISO26262 allows for the sampling of the fault list [4], facilitating a more efficient evaluation process. Nevertheless, applying a sampling factor must comply with stringent criteria, such as confidence levels and the specific nature of safety mechanisms, which means that the fault simulation time will still carry a certain burden. This is further amplified by the extended duration needed to obtain more precise information through the simulation of a full fault list. Consequently, expediting the fault simulation process is crucial to enhancing the efficiency and adaptability of functional safety verification.

A lot of research has been done in the area of fault simulation acceleration. PROOFS [5] identifies inactive faults in each time frame and only simulates the active faults. However, all state nodes with faulty values must be stored to identify inactive faults in the next time frame, resulting in significant overhead when applied to functional patterns due to the large number of clock cycles involved. HOPE [6] further reduces the faults to be simulated in the fanout-free region but requires additional local logic and fault simulations to map non-stem faults and examine stem faults in each time frame, which makes it impractical for functional patterns with numerous clock cycles. Other improved methods have also been proposed [7] [8], but none consider the characteristics of functional patterns, making them unsuitable, while some studies focus on functional patterns for fault grading at the RTL [9] [10] but fail to provide accurate fault coverage. In addition, some researchers turned to emerging technologies, such as parallel computing with multi-core CPUs and GPUs [11] [12] [13], to speed up fault simulation for scan patterns. As far as we know, acceleration techniques for fault simulation presented in the existing literature primarily target scan patterns while there is a notable lack of acceleration

978-3-9826741-0-0/DATE25/© 2025 EDAA

methods for functional patterns. This leaves the speeding up of fault simulation for functional patterns as a critical challenge yet to be addressed.

In the paper, we propose a parallel fault simulator to achieve efficient fault simulation for functional patterns. With the characteristics of functional patterns in mind, three techniques are deployed in the fault simulator to accelerate the fault simulation process. 1) Fault filtering is conducted to decrease the number of faults requiring simulation. 2) Fault grouping is performed to group faults with an early stop time frame and cease the simulation process ahead of time. 3) The implementation of CPU-based parallelism, in harmony with the NUMA architecture, maximizes the performance of our fault simulator.

The remaining paper is organized as follows. Backgrounds are described in Section II. Section III presents our proposed fault simulator. It is followed by experimental results and analysis in Section IV. Finally, the paper concludes with Section V.

II. Background

In this section, terms about fault simulation, fault parallelism and the NUMA architecture are introduced. Throughout this paper, we consider three logic values for signal lines: 0, 1, and *X*, where *X* represents an unknown state. We only assume the presence of single stuck-at faults within circuits.

A. Terms about Fault Simulation

Let's assume there is a stuck-at-*b* (*b* is 0 or 1) fault at line *L*, and its good machine value is *a* (*a* is 0 or 1 or *X*), then the fault effect at line *L* is expressed as *a*/*b*. If *a* and *b* are opposite, the fault is activated, otherwise, the fault is not activated.

A fault is detected if the fault effect manifests as a 0/1 or 1/0 difference at any of the circuit's outputs. If the only observable discrepancy is 0/*X* or 1/*X*, the fault is potentially detected. Conversely, the fault is considered to be undetected if the *X*/0 or *X*/1 difference is the only possible outcome.

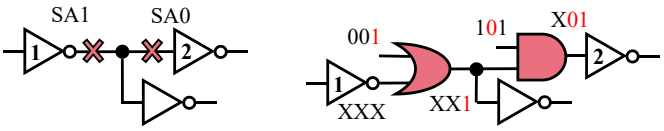
Due to the presence of flip-flops, faults at a time frame can be categorized into two types based on the origin of the fault effect [6]. The first scenario involves the fault effect emanating only from the fault site, classifying it as a single-event fault. In the second scenario, the fault effect arises not just at the fault site but also from some flip-flops, leading to its classification as a multiple-event fault. A multiple-event fault occurs when the fault effect reaches some flip-flops as a result of the input stimuli applied at the previous time frame.

B. Fault Parallelism

Accelerating fault simulation through bit parallelism involves two methods: pattern parallelism and fault parallelism. For scan patterns, each pattern lasts for about a couple of clock cycles at most, and they are independent of each other, making them highly suitable for pattern parallelism. In contrast, functional patterns span tens of thousands to millions of clock cycles, with subsequent cycles relying on the outcomes of preceding ones, rendering the application of pattern parallelism unrealistic.

However, when considering single stuck-at faults, each fault can be treated as independent from the others. Therefore, fault parallelism serves as a viable technique to accelerate fault simulation for functional patterns. Fault parallelism uses *w*-bit-wide data words to represent simulation values, which indicates that the number of parallel faults depends on the word length. When simulating a group of parallel faults, fault dropping is prohibited. All faults in the group must be simulated until the decision has been made for all the faults.

In fault parallelism, fault injection is one of the key aspects. One of the typical methods is modifying the circuit [5]. A two-input OR gate is inserted when injecting a stuck-at-1 (SA1) fault at a signal line, while a stuck-at-0 (SA0) fault can be injected by inserting a two-input AND gate. One of the inputs is used to determine the bit position of the inserted fault. Fig. 1 presents a simple example. The main overhead of this method lies in the need to repeatedly modify and recover the circuit. However, once the circuit modification is done, fault simulation will proceed as swiftly as logic simulation. Another method involves injecting faults by defining a fault descriptor [6], which avoids altering the circuit but introduces additional judgments and processing during the simulation process.



(a) Original Circuit (b) Modified Circuit

Fig. 1. Example of fault injection by modifying the circuit

C. NUMA Architecture

NUMA (Non-Uniform Memory Access) architecture is a memory organization used in multi-processor systems. It aims to scale the performance of high-end servers by minimizing the bottleneck of centralized memory access.

In the NUMA architecture, CPUs are divided into multiple NUMA nodes, each with its own independent memory space, as shown in Fig. 2. Under NUMA, a CPU accesses its own local memory faster than non-local memory (memory local to another node) as it does not need to traverse the interconnect first [14]. According to this characteristic, [13] proposed a method to accelerate fault simulation for scan patterns. The approach includes allocating private data in local memory and replicating thread-shared public data, thereby helping to reduce memory access latency.

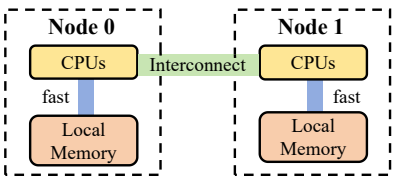


Fig. 2. Schematic diagram of NUMA architecture

III. Proposed Fault Simulator

In this section, an overview of our proposed fault simulator for functional patterns is described. After that, our three

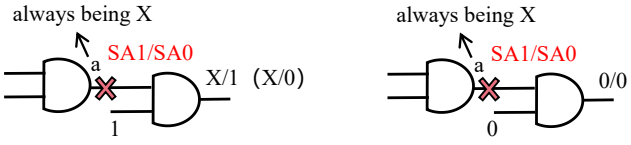
proposed techniques, which substantially reduce the fault simulation time, are presented: 1) fault filtering, 2) fault grouping, and 3) CPU-based parallelism.

A. Overview of the Simulator

Our proposed fault simulator is implemented based on fault parallelism. At first, a one-time good machine simulation based on the whole functional pattern is conducted to collect useful information. Based on this information, fault filtering and fault grouping are then performed. After that, we choose to inject the fault by modifying the circuit. Given that functional patterns may span millions of clock cycles, investing some effort in modifying and recovering the circuit before and after simulation can avoid the need for numerous additional operations throughout the simulation process. After faults are injected, the simulation is executed in parallel by multiple threads. Each thread processes the faults in the order of time frames. Within each time frame, the input stimuli are first applied to the circuit's inputs, followed by the event-driven simulation. The detection of faults is then determined by comparing whether the circuit's output responses match the expected output responses. The simulator will keep working until all the faults in the fault list have been processed.

B. Fault Filtering

A new method is proposed that filters the fault list to reduce the number of faults requiring simulation. Since good machine values of some gates will persist as *X* under the given functional pattern, certain faults will remain undetectable. This is because their fault effects are only present in two scenarios during the fault simulation process. One remains as *X*/0 or *X*/1, as shown in Fig. 3(a). The other is converted to 1/1 or 0/0 by other input ports, as illustrated in Fig. 3(b). Both of these fault effects are incapable of making the fault detected or potentially detected.

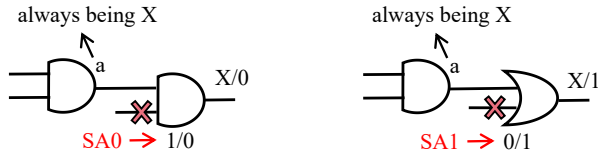


(a) Scenario 1 (b) Scenario 2

Fig. 3. Example of fault effect propagation (faults at gate with good machine value remaining *X*, where *X* comes from some uninitialized flip-flops)

It should be noted that faults at other inputs of their successors may also be undetectable. Fig. 4 gives two examples. If the successor is an AND gate, the SA0 fault at another input will remain undetected, while the SA1 fault will remain undetected if the successor is an OR gate. Generally, whether a fault is considered undetectable depends on the gate type of the successor.

Our strategy is to find out these faults and remove them from the fault list. First, gates whose values remain *X* during the whole good machine simulation process will be collected. Then, faults at these gates' output will be removed from the fault list. Based on the gate type of successor, faults deemed undetectable will also be removed. Only the remaining faults in the fault list will be used for fault simulation.



(a) Example 1 (b) Example 2

Fig. 4. Examples of fault effect propagation (faults at other input of successor)

C. Fault Grouping

The one-time good machine simulation also reveals the last change in the good machine values of gates. This information can assist in grouping the faults remaining after the filtering process. The motivation is shown in Fig. 5. As the functional pattern is sequential, the fault simulation can be viewed as occurring along a timeline. Let's assume there is a gate whose good machine value remains unchanged after time frame *n*. If we can identify certain faults whose fault simulation can be ceased at that specific time frame without changing the result, the time required for simulating these faults could be significantly reduced.

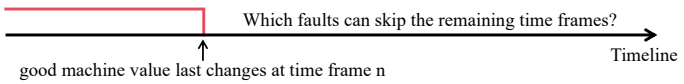
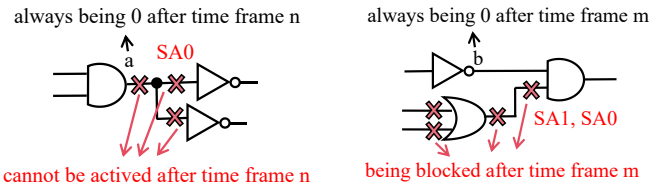


Fig. 5. Motivation for fault grouping

Based on this consideration, two types of faults are identified. One type includes faults that cannot be activated by the last good machine value, and the other includes faults that will be blocked by that value. Fig. 6 provides examples of both types. As shown in Fig. 6(a), the SA0 fault at the output of the AND gate cannot be activated after time frame *n* because the good machine value of the AND gate remains 0 after that point. Also, SA0 faults at the input of the two inverters cannot be activated after time frame *n* since they are driven by the AND gate. In Fig. 6(b), with the inverter's good machine value persistently at the controlling value 0 after time frame *m*, both SA1 and SA0 faults at the other input of the successor (the AND gate) are blocked after then. Moreover, all of the faults at the inputs and output of the OR gate are all blocked after time frame *m* since there is no other path to propagate their fault effect.



(a) Example 1 (b) Example 2

Fig. 6. Examples of faults that cannot be activated and faults that are blocked by the last good machine value

In our method, the above faults will be identified and tagged. The tag indicates at which time frame the fault simulation for

the fault can be stopped. On the contrary, faults not tagged will be simulated to the final time frame.

After tagging these faults, fault grouping is performed. Firstly, faults will be extracted from the fault list. Secondly, faults with the same stop time frame will be grouped together. Thirdly, the stop time frames will be sorted in ascending order. Fourthly, starting from the smallest time frame, faults that stop at that time frame will be added back to the fault list. After adding faults of the current time frame, move on to the next time frame until faults with tags are all added. Finally, faults that should be simulated to the final time frame will be added back to the fault list. A simple example is presented in Fig. 7.

In Fig. 7, the yellow array at the top represents the fault list before fault grouping, with its eight elements representing eight faults (named by *f0* to *f7*). Faults written in italics indicate that the faults are tagged. After extracting and grouping the faults, they are divided into four groups, as illustrated by the four dashed rectangles in the middle. The first dashed rectangle includes *f0* and *f6*, indicating that they can be stopped at frame 2, and so on. After grouping the faults, they are added back to the fault list in ascending order of time frames. The fault list after fault grouping is shown as the yellow array at the bottom.

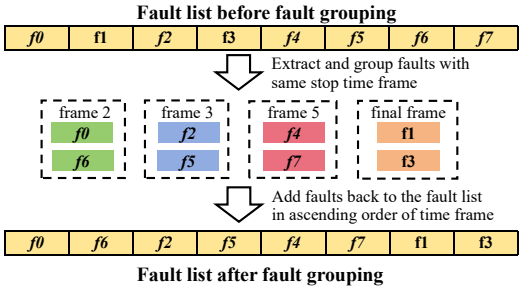


Fig. 7. Fault grouping with stop time frames

Since the fault simulation is conducted in a bit-parallel fault manner, faults assigned to the same batch can only be halted simultaneously. To ensure the accuracy of every fault's result, the simulation for all faults within a batch must end at the time frame corresponding to the last fault.

It should be highlighted that even if faults are tagged to stop at a certain time frame, some may require simulation until the final time frame for correct results. This is due to the presence of flip-flops. Although a fault becomes inactive or unable to propagate its effect after a certain time frame, it could still be detected later if the fault effect is retained in any flip-flop. This means that if a fault transitions from a single-event fault to a multiple-event fault before the stop time frame arrives, it should simulate to the final time frame rather than stop earlier.

However, simulating an entire batch of faults, which are tagged to stop earlier, to the final time frame due to just a few multiple-event faults leads to significant inefficiency. Therefore, our proposed method employs a specific strategy. The simulation will continue to the final time frame only when the stop time frame (n) and the number (m) of multiple-event faults exceed the empirical thresholds. The thresholds are determined through experiments and set at four levels: (1) $n > 3/4\ N$, (2) $n > 1/2\ N$ and $m > 1/2\ M$, (3) $n > 1/4\ N$ and $m > 4/5\ M$, (4) $m = M$, where N and M denote the number of time frames and the number of faults in the batch, respectively. If none of these conditions is met, the simulation will still stop at time frame n. To ensure the final correctness of the results in this case, multiple-event faults will be collected and preserved until the entire fault list is processed, after which they will be simulated again. Since the number of multiple-event faults requiring a second simulation under this strategy is not overly large, the overhead is deemed acceptable. An example to illustrate this strategy is provided in Fig. 8. Faults written in italics indicate that the faults can be stopped at time frame n. Conversely, they transition to non-italic when they need to simulate to the final time frame.

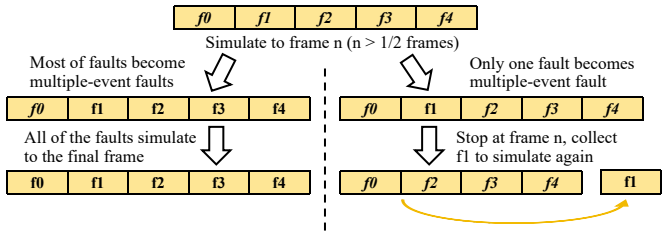


Fig. 8. Stop/continue the simulation when multiple-event faults appear

It is worth noting that the information about the first change in the good simulation values of gates can also be used for fault grouping. However, in practical applications, it does not yield significant benefits. Therefore, in the proposed fault grouping technique, only the information about the last change in the good simulation values of gates is considered.

D. CPU-based parallelism

After grouping the faults, fault simulation is performed. In our proposed fault simulator, some CPU-based parallel technologies are used to accelerate the simulation process.

As the fault simulation process is implemented based on fault parallelism, the number of parallel faults is decided by the word length. In our proposed fault simulator, we use X86_64 AVX2 intrinsics to extend the simulation value to 256 bits as shown in Fig. 9. Since three-value logic (0, 1, X) is considered, two 256-bit integers are used for one gate, with each corresponding bit pair representing a simulation value. A simple coding scheme is used, where 00, 11, and 10 denote the values 0, 1, and X, respectively. In each of the two integers, one bit out of 256 is dedicated to representing the fault-free circuit, with the remaining 255 bits designated for the 255 faulty circuits, which means the number of parallel faults can reach up to 255. This strategy is informed by the potential for functional patterns to extend over vast numbers of clock cycles, where recording good machine value of every time frame in advance would necessitate prohibitive memory use.

Moreover, fault simulation is expedited by employing multi-threading via the OpenMP API. For reducing memory access latency under NUMA, the characteristics of the NUMA architecture are taken into account. The LIBNUMA Linux library is used to bind threads and allocate memory to NUMA nodes.

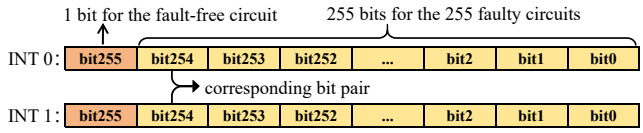


Fig. 9. Simulation value of one gate using AVX2 intrinsics

In our proposed method, threads work on different faults in parallel by making use of data parallelism. At first, threads are launched and evenly bind to different NUMA nodes. Then, the private data of threads, which include simulation values of gates and event queue for event-driven simulation, will be allocated to the local memory of the NUMA node to reduce memory access latency. While public data like netlist and fault list, are handled in different ways. Since faults will be injected by modifying the netlist, multiple copies of the netlist are made. One copy of the netlist is provided for each thread and allocated to the local memory. The way to bind threads and allocate data is presented in Fig. 10. As for the fault list, rather than dividing it into several parts before the simulation as [13], it will be accessed as a whole by all threads for allocating faults dynamically.

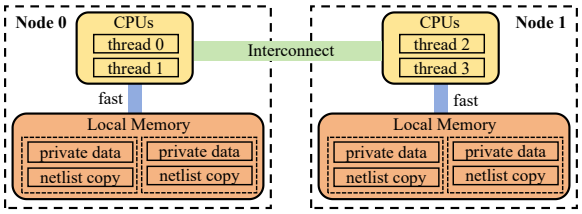


Fig. 10. Threads binding and data allocation

At the beginning of the simulation process, each thread will load a group of consecutive faults from the fault list in sequential order. The faults loaded to each thread will be executed through fault parallelism. As threads will work on their own copy of the netlist, once a thread completes the current group of faults, it can immediately recover the netlist and modify it for the next group of faults without waiting for other threads. Therefore, each thread can be regarded as a "long-lived" thread since it will keep working until no faults are left. This arrangement naturally achieves a balanced distribution of workload among the threads.

In this parallel approach, the only potential for a race condition arises when two threads attempt to load a new group of faults simultaneously. However, this can be avoided with atomic operations. As different threads are working on different groups of faults, the probability of simultaneous completion by two threads is remarkably low. Therefore, the overhead resulting from atomic operations is considered acceptable.

The pseudo-code of the fault simulation function is provided in Algorithm 1. The outermost ***for*** loop will execute in parallel under the effect of the OpenMP pragma statement. Each parallel thread performs fault simulation by sequentially processing groups of faults through the execution of a ***while*** loop. Once all the faults in the fault list have been processed, the fault simulation finishes.

Algorithm 1 Fault simulation by using multi-threading

Input: fault list; functional pattern
Output: fault simulation result

```
1:  #pragma omp parallel for
2:  for every thread T do
3:      bind thread T and allocate data on the NUMA node;
4:      while true do
5:          #pragma omp atomic
6:          get a group of faults;
7:          if all faults in the fault list have been processed then
8:              break;
9:          end if
10:         inject faults by modifying its own netlist copy;
11:         event-driven fault simulation in every time frame;
12:         remove faults by recovering its own netlist copy;
13:     end while
14: end for
```

TABLE I
ATTRIBUTE OF EXPERIMENTAL CIRCUITS

Design	#Gates	#PIs	#POs	#FFs	#Faults	#Cycles
pci_bridge32	38,653	162	107	3,359	122,880	50K
wb_conmax	66,605	1,130	1,416	770	223,042	50K
ethernet	107,665	96	115	10,544	367,978	50K
RISC	128,395	276	351	7,391	425,154	50K
vga_lcd	245,256	89	109	1,7079	836,916	50K
design1	152,162	40	42	15,737	540,040	55K
design2	164,136	40	42	16,225	594,354	125K
design3	207,852	40	42	18,388	730,118	1M

IV. EXPERIMENTAL RESULTS AND ANALYSIS

Experiments are performed on a Linux workstation with dual-socket processors. The processor model is Intel Xeon Gold 6252@2.10GHz. There are two NUMA nodes. Each node corresponds to a processor with 24 cores and 64 GB of local memory, totaling 48 cores and 128GB of memory overall.

Experimental data are collected with five open-source designs from IWLS05 benchmarks [15], as well as three CPU core designs named design1, design2, and design3. The attributes of the designs are presented in Table I. The pattern lengths of functional patterns are also listed, which are calculated in terms of the number of clock cycles. The functional patterns used for the three CPU core designs are generated by the design testbench during their functional verification, whereas the patterns for the five benchmark circuits are randomly generated. These patterns are stored in files with VCD (Value Change Dump) format.

For comparison, the single-threaded version and multi-threaded version of our fault simulator are realized. The single-threaded version only utilizes 64-bit parallelism technology, while the multi-threaded version incorporates our three proposed techniques. A commercial tool, Synopsys's TestMAX, is also used to run fault simulation for functional patterns. It should be noted that running fault simulation for functional patterns with multi-threading is not supported in the commercial tool. Larger cases such as multi-million-gate designs are not used in the experiments because the runtime of the single-threaded version and the commercial tool would be prohibitively long (more than several months), making data

TABLE II
RUNTIME AND SPEEDUP DATA

Design	Runtime (sec.)					Speedup (x) vs. Commercial tool			
	Commercial tool	Single-threaded	Parallelism	Grouping + Parallelism	Filtering + Grouping + Parallelism	Single-threaded	Parallelism	Grouping + Parallelism	Filtering + Grouping + Parallelism
pci_bridge	19,385.23	12,383.66	179.75	169.14	134.55	1.57	107.85	114.61	144.07
wb_conmax	19,156.25	55,045.37	1,221.19	1,081.90	1,081.90	0.35	15.69	17.71	17.71
ethernet	115,245.58	82,271.43	1,532.92	1,543.62	849.87	1.40	75.18	74.66	135.60
RISC	52,753.27	47,065.93	952.05	740.34	717.08	1.12	49.44	71.26	73.57
vga_lcd	240,902.80	418,463.22	9,713.26	9,562.36	6,815.80	0.58	24.80	25.19	35.34
design1	650,113.85	582,703.15	11,435.08	10,155.79	8,158.91	1.12	56.85	64.01	79.68
design2	1,643,539.09	1,448,926.53	30,013.65	26,857.63	22,372.21	1.13	54.76	61.19	73.46
design3	several months	several months	4.2 days	3.8 days	3.3 days	n/a	n/a	n/a	n/a
						Ave. 1.04	**Ave.** 54.94	**Ave.** 61.23	**Ave.** 79.92

collection difficult. Besides that, we do not compare our work with other parallel fault simulation methods [11] [12] [13] because they do not support fault simulation for functional patterns, which is a key contribution of our work.

Table II presents the runtime and speedup data. Columns 2 and 3 list the runtime data of the commercial tool and our single-threaded version respectively. Column 4 shows the outcomes achieved by CPU-based parallelism (Parallelism). Column 5 presents the result garnered from integrating fault grouping with CPU-based parallelism (Grouping + Parallelism). Column 6 lists the runtime data of our fault simulator using all three techniques (Filtering + Grouping + Parallelism). The speedup results are listed under column 7 to column 10. In the experiments, the number of parallel threads is set to 48.

The speedup data are computed against the commercial tool. With the gradual incorporation of the three techniques, the average speedup reaches 54.94x, then escalates to 61.23x, and finally advances to 79.92x, about 1.11x and 1.31x performance increase. This indicates that each of the three techniques plays a significant role in improving performance. Due to the page limit, the runtime data for different thread counts, which show that our parallel algorithm has good scalability, are not presented. In addition, the speedup data for *wb_conmax* and *vga_lcd* are lower than those of the others, likely due to different algorithms used in the commercial tool.

Table III presents the experimental data of our proposed fault grouping and fault filtering techniques. Column 2 lists the number of multiple-event faults that are simulated twice because of the fault grouping technique. As we can see, these percentages are relatively low (no more than 3%), indicating that the overhead caused by simulating some multiple-event faults twice is acceptable. Column 3 lists the number of faults that are filtered out because of the fault filtering technique. These percentages are case-sensitive as they depend heavily on gates whose good machine values persist as *X* under the given functional pattern. For example, the percentage data of *ethernet* is more than 30%, while no fault can be filtered in *wb_conmax*. This is due to the fact that there are many PIs (more than 1k) and very few FFs (only 1.15% of the total gates), making the circuit easy to control and causing the good machine values of most of the gates to change to deterministic values rather than persist as *X*. The last column presents the elapsed time of the preprocessing process (including the one-time good machine simulation), which is incorporated in the total runtime listed in Table II. As is shown, the overhead introduced by the preprocessing process constitutes only a small portion of the total runtime but yields significant benefits.

TABLE III
EXPERIMENTAL DATA OF FAULT GROUPING AND FILTERING

Design	#Faults Sim Twice	#Faults Filtered Out	Preprocessing Time (sec.)
pci_bridge32	1,559 (1.3%)	16,869 (13.7%)	8.6 (6.4%)
wb_conmax	781 (0.4%)	0 (0.0%)	34.2 (3.2%)
ethernet	361 (0.1%)	111,845 (30.4%)	18.4 (2.2%)
RISC	8,322 (3.0%)	7,966 (1.9%)	11.5 (1.6%)
vga_lcd	2,420 (0.3%)	113,725 (13.6%)	60.5 (0.9%)
design1	13,446 (2.5%)	56,309 (10.4%)	97.2 (1.2%)
design2	12,727 (2.1%)	51,059 (8.6%)	236.3 (1.1%)
design3	12,007 (1.7%)	49,559 (6.8%)	2176.4 (0.8%)

V. CONCLUSION

In the paper, an efficient parallel fault simulator for functional patterns on multi-core systems is proposed. With the characteristics of functional patterns in mind, three techniques are deployed in the fault simulator to reduce the runtime of fault simulation from different perspectives. 1) To reduce the number of faults to be simulated, fault filtering is performed. 2) Faults eligible for early termination are grouped together, allowing the simulation for them to conclude ahead of schedule. 3) The deployment of CPU-based parallelism, coupled with the NUMA architecture, further accelerates the simulation speed. By employing these three techniques, an average 79x speedup can be obtained on a 48-core system.

REFERENCES

[1] R. H. Leo, Why Functional Safety in Road Vehicles?, Chan: Springer International Publishing, 2016, pp. 7-39.

[2] F. A. da Silva, et al., "Combining fault analysis technologies for ISO26262 functional safety verification," IEEE Asian Test Symposium, 2019, pp. 1290-1295.

[3] L. Lai, et al., "Parallel Logic Simulation for Functional Test," Journal of Computer-Aided Design & Computer Graphics (in Chinese). 2023, pp. 803-810.

[4] ISO: ISO 26262 Road Vehicles - Functional Safety, ISO Standard, 2018.

[5] T. M. Niermann, et al., "PROOFS: a fast, memory-efficient sequential circuit fault simulator," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1992, 198-207.

[6] H. Lee, D. Ha, "HOPE: An efficient parallel fault simulator for synchronous sequential circuits," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1996, pp. 1048-1058.
[7] C. R. Graham, E.. M. Rudnick and J. H. Patel, "Dynamic fault grouping for PROOFS: a win for large sequential circuits," International Conference on VLSI Design, 1997, pp. 542-544.
[8] C. Kung, C Lin, "HyHOPE: fast fault simulator with efficient simulation of hypertrophic faults," IEEE International Conference on Computer-Aided Design, 1994, pp. 53-60.
[9] P. A. Thaker, V. D. Agrawal, M. E. Zaghloul, "Register-transfer level fault modeling and test evaluation techniques for VLSI circuits," IEEE International Test Conference, 2000, pp. 940-949.
[10] H. Fang, et al., "RT-Level Deviation-Based Grading of Functional Test Sequences," IEEE VLSI Test Symposium, 2009, pp. 264-269.
[11] H. Li, et al., "nGFSIM: A GPU-based fault simulator for 1-to-n detection and its applications," IEEE International Test Conference, 2010, pp. 1-10.
[12] J. Hu, et al., "Adaptive Multidimensional Parallel Fault Simulation Framework on Heterogeneous System," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2022, pp. 1951-1964.
[13] S. Ye, et al., "Fault Simulation Acceleration Based on ARM Multi-core CPU Architecture," IEEE Asian Test Symposium, 2023, pp. 1-5.
[14] C. Lameter, "An overview of non-uniform memory access," Communications of the ACM, 2013, pp. 59-64.
[15] IWLS 2005 Benchmarks. 2005. http://iwls.org/iwls2005/benchmarks.html.