CAD Group

RESCUE
European Training Network

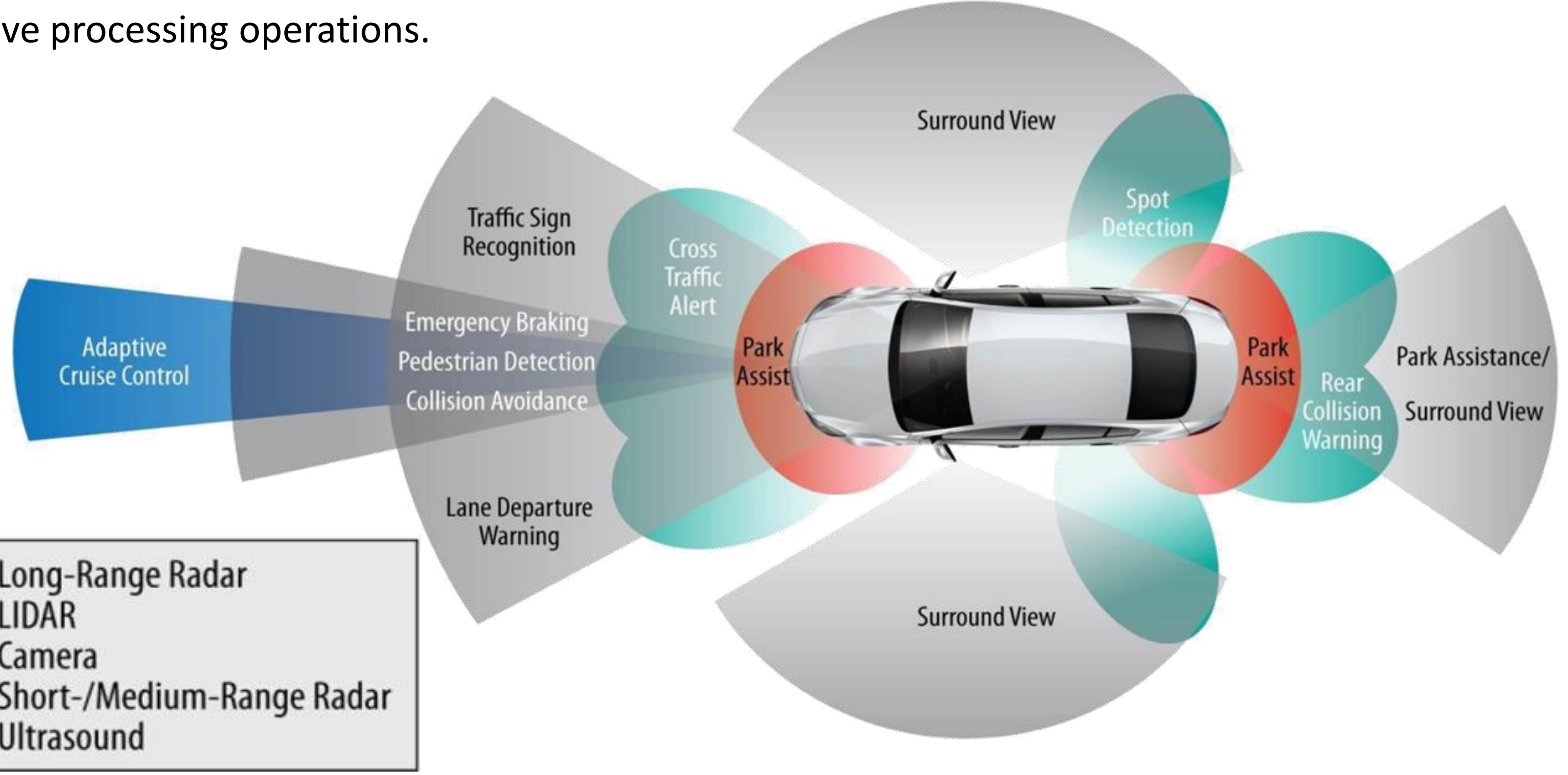
H2020 Programme, under grant n. 722325.

IN-FIELD GPGPU TEST WITH SBST TECHNIQUES

B. Du*, Josie E. Rodriguez Condia†, M. Sonza Reorda‡, L. Sterpone§
Electronic CAD & Reliability Group (CAD)
Politecnico di Torino, Torino, Italy
{*boyang.du, †josie.rodriguez, ‡matteo.sonzareorda, §luca.sterpone}@polito.it

INTRODUCTION

General Purpose Graphical Processing Units (GPGPUs) are increasingly used as effective solutions in safety critical applications such as the automotive ones due to their capabilities in data intensive processing operations.



In these field, the GPGPUs must match a set of safety standards to guarantee the correct in-field operation (**ISO26262, IEC 61508**).

These regulations include the requirements of functional safety of electronic systems and correct execution of internal modules (**Safety, Reliability**).

Requirements are not easily evaluated during in-field operation. Hence, techniques are required to test them during in-field operation with respect to possible permanent faults arising when the device is already deployed in the field.

Motivation

We aim first analyzing the effects of permanent faults in the GPGPU operation.
(example)


Original Image.


Edge detection with Sobel filter.


Edge detection result with a permanent fault in SM0 actual mask field (Thread 5), 8 threads per block.

Secondly, we aim at developing effective **Software-based Self-Test(SBST)** [1] techniques in presence of permanent faults.

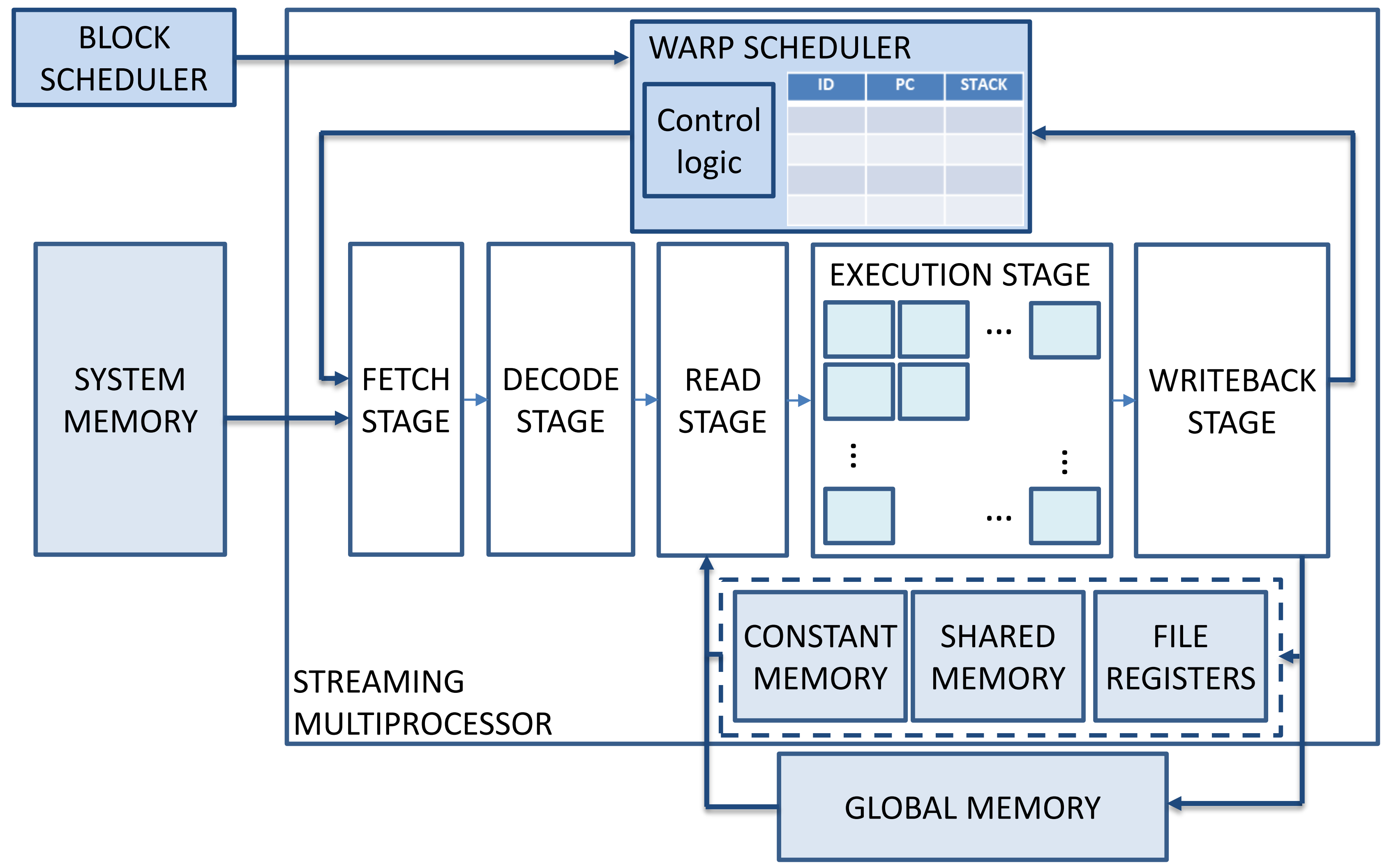
SBST are software routines developed to verify the integrity of internal modules of a system.

- Based on architectural or structural description of modules (**Self-test program**).
- Advantages by **at-speed** and **in-field** testing (**functional testing**).
- Non-intrusiveness
- Flexibility
- Test duration

The development of the functional test code addressing the several computational cores composing a GPGPU can be done resorting to known methods developed for CPUs [2], for other modules which are typical of a GPGPU we still miss effective solutions [3-5] . This work focuses on the scheduler unit which is in charge of managing different scalar computational cores and the different executed threads.

At first, we propose SBST methods for evaluating the fault coverage that can be achieved using an application program. Then, we provide some guidelines for improving the achieved fault coverage. Experimental results are provided on an open-source VHDL model of a GPGPU [6].

FLEXGRIP GPGPU ARCHITECTURE


BLOCK SCHEDULER

WARP SCHEDULER

Control logic

ID | PC | STACK

SYSTEM MEMORY

FETCH STAGE

DECODE STAGE

READ STAGE

EXECUTION STAGE

WRITEBACK STAGE

CONSTANT MEMORY

SHARED MEMORY

FILE REGISTERS

STREAMING MULTIPROCESSOR

GLOBAL MEMORY

PROPOSED METHODS

The methods are based on the memory line entry description for the models.
Entry line:

Warp. STATE	BLOCK CONFIG	Warp. PC	Warp Actual. MASK

METHOD	M1	M2	M3
FIELDS	Warp Actual. MASK Warp. PC		
BASED ON	- Thread Divergence. - Thread Routine placement in system memory.	- Bottleneck of global memory storage. - Mixing the application code with the SBST method.	- M2 method with a Thread signature storage.
ADVANTAGE	- Faults identified by Performance degradation **(performance counters)** and System hanging.	- Performance degradation **(performance counters)**. - Memory mismatch **(memory content observability).**	- Use of results in memory to verify the operation of module **(memory content observability).**

General Pseudo-code to describe the proposed algorithms to test the scheduler Memory.

```
j ← 0                                          ▶ Clear constant
…                                              ▶ Normal app. Execution
Sig_per_thread[] ← 0                           ▶ Initialize signature      (M3)
for i ϵ {set of ThreadId in SM} do             ▶ Evaluate for every ThreadID
    if i == j then                             ▶ If ThreadID Matches
        Divergence_path_GroupA();              ▶ Divergence path Group A
        NOP                                    ▶ Not operation instruction
        Thread_Store_in_memory();              ▶ Memory results store      (M2)
        Sig_per_thread[i] ← Sig_per_thread[i]+1 ▶ Set signature            (M3)
        Sig_store_in_memory();                 ▶ Store signature           (M3)
    else
        Divergence_path_GroupB ();             ▶ Divergence path Group B
    j ←j+1                                     ▶ Change constant value
```

.SASS

```
…                     ▶ Application code
GLD Rx, g[0x06]       ▶ Move of threadIdx.x (stored in shared memory)
MVI Ry, Z             ▶ Move constant parameter per SP (from 0 to (Z-1))
…                     ▶ Application code
TEST_N:
AND Rx, Ry            ▶ Comparison (Z) and threadIdx.x
SSY Dir_1             ▶ Convergence point definition
BRANCH Dir_2          ▶ Conditional evaluation
NOP                   ▶ Divergence Path
NOP
Dir_2:GST M[Ra],Rb    ▶ Convergence Path, Storage thread results
Dir_1:  NOP.S         ▶ Warp branch stack release (Convergence point)
--- Repeat Z-1 times according to the number of threads per block.
```

CUDA C

```
…                        ▶ Normal application code
switch(threadIdx.x)      ▶ Comparison of threadIdx.x
{
case Z:                  ▶ Thrd. execution for threadIdx.x = Z
Thread_final_Store();    ▶ Store of results in global memory
break;
…                        ▶ Comparison with other Z-1 value
}                        ▶ End of M2 code
```

EVALUATION

Model simulation in: ModelSim
Experimental results:

Warp scheduler memory (Pool/Stack)

Application Code	VectorAdd	M1	M2	M3
Total Faults	2,048	2,048	2,048	2,048
Testable Faults	984	984	984	984
Detected Faults	624	728	984	984
Hang	440	**613**	616	616
Memory Mismatch	184	115	112	**368**
Performance degradation	0	0	**256**	0
Testable Fault coverage (%)	63.41	73.98	100	100
Fault coverage (%)	30.46	35.54	**48.04**	**48.04**

Interconections (WARP Scheduler - Shader)

Application Code	VectorAdd	M1	M2	M3
Total Fault	478	478	478	478
Testable Faults	277	277	277	277
Detected Faults	155	177	238	236
Hang	105	157	154	161
Memory Mismatch	50	20	20	**75**
Performance degradation	0	0	**64**	0
Testable Fault Coverage (%)	55.95	63.89	85.92	85.20
Fault Coverage (%)	32.42	37.02	**49.79**	49.37

M1 is able to detect some faults; however, the fault coverage is low.

M2 achieves higher fault coverage by introducing store instruction to access global memory to increase performance variation among different divergence paths.

M3 achieves similarly high fault coverage by only checking the final results in global memory, taking advantage of a signature variable for each thread.

Conclusions:

- First, we experimentally proved the serious effects that permanent faults in the scheduler may cause.
- The key idea of proposed methods is their capability to generate divergence paths of thread execution and use performance variation among the threads and/or final results in global memory to detect permanent faults.
- Fault injection campaigns have been carried out using FlexGrip. Results indicate that both method M2 and M3 are promising SBST methods able to achieve high fault coverage.
- The M3 method requires only to check the final results in memory after test program execution, which is a typical mechanism used in processor SBST techniques.

Future works:

- To extend the characterization to further GPGPU modules and to compare the fault coverage results with extended Instruction Set Architecture (ISA) fault simulators.
- To use the proposed techniques on gate-level netlist models and real GPGPU embedded platforms.

REFERENCES

[1] Stefano Di Carlo; Giulio Gambardella; Marco Indaco; Ippazio Martella; Paolo Prinetto; Daniele Rolfo; Pascal Trotta, “A software-based self test of CUDA Fermi GPUs”, 2013 18th IEEE European Test Symposium (ETS)

[2] A. Paschalis; D. Gizopoulos, “Effective software-based self-test strategies for on-line periodic testing of embedded processors”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Year: 2005, Volume: 24, Issue: 1, Pages: 88 – 99

[3] N. Farazmand, R. Ubal, and D. Kaeli, “Statistical fault injectionbased AVF analysis of a GPU architecture,” in Proc. 8th IEEE Workshop Silicon Errors Logic Syst. Effects, Year: 2012.

[4] J. Tan, Y. Yi, F. Shen, and X. Fu, “Modeling and characterizing GPGPU reliability in the presence of soft errors,” Parallel Comput., Year: 2013, vol. 39, no. 9, Pages 520–532

[5] David Defour, Eric Petit, “A software scheduling solution to avoid corrupted units on GPUs”, Journal of Parallel and Distributed Computing, Year: 2016, Volumes 90–91, Pages 1-8

[6] Kevin Andryc; Murtaza Merchant; Russell Tessier, “FlexGrip: A soft GPGPU for FPGAs”, 2013 International Conference on Field-Programmable Technology (FPT), pp. 230 – 237