POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Leveraging Automatic Test Equipment to optimize System-Level Test for Multicore Automotive SoCs

Supervisors
prof. Paolo Bernardi
prof. Stefano Quer
dott. Francesco Angione
dott. Lorenzo Cardone

Candidate
Lorenzo Bertetto

October 2024

Abstract

System-Level Test (SLT) is a relatively new test approach, performed on top of many other manufacturing test phases to enforce Automotive SoCs' reliability. SLT is applied by specific Automatic Test Equipment (ATE), which programs the SoC to run Functional Test Programs and monitors their execution to ensure the integrity of the SoC's functional capabilities. A current industrial concern is how to grade the ability of SLT procedures to stimulate the SoC resources. This work introduces an advanced technique tailored for multicore SoC architectures that leverages execution trace analysis of functional programs executed within an ATE ecosystem, including CPUs and FPGA resources. A comprehensive data flow graph is extracted from the execution traces on the ATE system to encapsulate all read/write operations on registers and memory addresses. This graph is then dynamically analyzed to assess whether each data point propagates correctly to the program termination. A custom metric is computed to quantify the overall data flow integrity. The method accounts for synchronization across multiple cores, ensuring accurate evaluation of data flow within and across logical processing units. Extensive experimentation on a STMicroelectronics automotive device demonstrates the method's computational efficiency and quantifies the gain in metric accuracy, time and human resources.

Summary

The first and main topic of the thesis is the extension of the existing methodology for computing the Connectivity. The Connectivity is a novel metric aimed at evaluating the usefulness of each instruction of the execution of a disassembly trace. In particular, the programs that are evaluated are System-Level Test (SLT) programs: the ones that are used to discover Hardware defects in the SoCs. The Connectivity algorithm analyzes each instruction and finds if there are some that do not contribute to the computation of the signature, which is the final output of SLT programs. The operations required to evaluate a SLT program are the following:

- The SLT program is executed on the target SoC with a debugger (ATE) that dumps each instruction and produces a disassembly trace of execution.
- The trace is transferred from the debugger to an external PC.
- The trace is then analyzed by the Connectivity algorithm, executed on the PC.

The extension focuses on the multicore evaluation. Before this extension, the Connectivity metric could only evaluate the trace of execution of a single CPU core. Because of this limitation, there were some cases where the Connectivity evaluation was inaccurate. For instance, no steps were done to check if a different core of the SoC read the data written in memory by an instruction. Similar situations are problematic since the original methodology did not track that data flow.

The dump of the traces and the computation of the Connectivity require a lot of time. The debugger slows down the execution of a program by orders of magnitude. Thirty seconds of real-time execution can become several hours when the debugger dumps the instructions. This time must be added to the time required to perform the evaluation once the trace is completely dumped. An entirely new approach has been proposed to speed up the computation. A new algorithm was developed to perform the evaluation in parallel with the dump of the instructions. The advantage of this new approach is that the time for performing the evaluation is reduced because the evaluation is completed as soon as the trace dump is completed. The evaluation is directly performed on the tester Hardware (ATE). Two main problems had to be faced when developing this algorithm:

- Space complexity (the tester has limited memory, so the algorithm must not consume too much memory)
- Synchronization between the debugger, which dumps one instruction at a time, and the algorithm, that must recompute the partial evaluation at each new instruction.

The multicore algorithm proved to increase the connectivity accuracy of multicore programs. Multicore programs were evaluated using the single-core and multicore versions of the algorithm to compare their Connectivity. The runtime algorithm was tested by running it on the tester

Hardware. Its low space complexity allowed it to execute without exceeding the memory limits. The synchronization between the algorithm and the debugger has been implemented correctly. Ideally, the algorithm should have been always faster than the debugger, but sporadically it is not, so the debugger must pause the execution waiting for the computation to finish in some cases. Despite this, a significant amount of time is still saved compared to the previous approach.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

System-Level Test (SLT) [10] has recently been introduced as an additional manufacturing test flow phase to intercept marginal defective behaviors that escaped previous test phases such as Scan and Built-In Self-Test test procedures aiming to test every system component separately. SLT is usually performed by specialized Automatic Test Equipment (ATE) [1] that uploads, launches, and monitors the execution of Functional Test Programs to verify the integrity of the SoC's functional capabilities. Nowadays, SLT challenges have become even more pronounced with the increasing complexity of multi-core SoC architectures.

A typical industrial practice for SLT often involves booting an Operating System and scheduling applications to mimic the mission behavior of the SoC. A key concern is how to effectively evaluate SLT programs' testing abilities. In particular, when chip makers create the SLT procedures, it is difficult to guess which mission applications will be programmed by chip users. Therefore, silicon test engineers struggle to create Functional Test Programs that must resemble the final application but also be broad enough in their functional scope to cover as many functional configurations as possible. Such a development process is often holistic, relying more on test engineers' experience than on computed coverage figures because it is challenging to set up fault coverage evaluation processes for a full System-on-Chip, and eventual measurement environments based on fault simulation tools are exceptionally time-consuming. In response, recent works [3, 2, 12, 9] have proposed indirect methodologies to quickly analyze the quality of functional test programs and grade their fault detection capabilities without fault simulation.

This work further explores indirect measurements as an alternative to fault simulation. In particular, it proposes to exploit the capabilities of the SLT-oriented ATE ecosystems to quickly grade the testing value of functional test procedures designed explicitly for SLT of multi-core architectures.

The proposed strategy leverages ATE architecture and its capabilities to retrieve execution traces from the SoC programmable elements, such as multiple CPUs inside the same SoC. The thesis illustrates how to compute a specific SLT metric, called connectivity [3], directly on-site at the ATE. Programmable elements of the ATE, such as microcontrollers and FPGA circuits, are used to crunch SLT information coming from the chip on the fly. By the direct analysis of instruction traces related to different CPUs and programmable SoC components, the ATE computes and returns connectivity values to the host PC controlling the SLT application instead of collecting and transmitting a large amount of data to be analyzed later, thus providing a substantial gain in time.

The algorithm described in the thesis permits a quick evaluation and direct validation of the test procedure on-board the ATE. It brings many significant benefits, including the possibility

of evaluation of SLT programs that tackle multi-core architectures and the early deployment of an SLT recipe for the ATE, which saves precious application engineer efforts.

The proposed methodology is applied to a medium-sized System-on-Chip (SoC) used in safety-critical applications in the automotive area. Experimental results show the evaluation of several System-Level Test applications developed on a ATE development station. The performance of the ATE-based method is compared with Fault Simulation and server-based trace analysis methods. Human effort benefits are qualitatively described, together with a description of a debug flow for SLT functional programs that enable climbing the connectivity metric by patching SLT, for example tuning compiler options.

The thesis is organized as follows. Chapter 2 introduces functional test programs and how to compute the connectivity metric offline and for single-core applications. Chapter 3 illustrates the methodology by introducing the multi-core connectivity and how to compute it run-time on board the ATE. Chapter 4 describes the case study and the experimental setups, and the results compared with previous techniques. Finally, Chapter 5 concludes the thesis with a summary and hints on possible future works.

Chapter 2

Background

2.1 Functional test program evaluation

The manufacturing test flow for SoCs usually includes phases ranging from structural methods (like Built-In Self-Test and other Scan-oriented techniques) to functional strategies (such as Software-Based Self-Tests). Such test strategies mainly deal separately with the individual test of the heterogeneous elements included in the SoCs (i.e., memories, logic gates, analog components, etc.).

For this reason, many silicon companies have also recently introduced an additional test phase, called System-Level Test (SLT) [4, 10, 14] to highlight problems due to heterogenous components interactions, communications, Hardware-Software interactions. SLT generally targets test escapes from previous test phases where DfT may also have introduced untestability [13, 7]. SLT's target is to run a certain application (often the one that will be shipped with the Hardware) on the target chip to test whether it works correctly without errors or failures. Unlike other test methodologies, SLT more directly considers the working conditions under which the Hardware will operate; thus, the SLT Software is often an operating system, i.e., the same that will be executed once the Hardware is shipped. Unfortunately, SLT is often unaware of the underlying Hardware structure and follows a "black box" approach. Consequently, its results are less correlated with the final fault coverage, whose reduction is the ultimate design target. Moreover, designing the test programs is often intrinsically problematic and requires repeated attempts, including time-consuming simulation and fault simulation campaigns.

2.2 Indirect measurements for SLT and Connectivity metric

Looking for alternative measurements of SLT effectiveness without simulation or fault simulation is a relevant topic today in the test community. These approaches include on-chip current measurement-based approaches like [11], and application trace-based ones at the microarchitectural level [9] or at low, Hardware-level, such as connecting to the real chip through a physical test port [3].

In the present thesis, Hardware-level traces are the source of information. An important background for the proposed strategy is the alternative metrics presented recently, i.e., the so-called connectivity [3]. This metric shows a significant correlation with fault coverage. Connectivity is a metric that analyzes the execution trace of a functional application and provides a fast way to

measure the effectiveness of an application in terms of its testing ability without the necessity of elaborating on the netlist design.

The central concept of connectivity is determining whether the generated data is accurately propagated to the program's endpoint or a designated signature point. Once this verification is complete, the strategy assigns a score that reflects the program's overall "connection" level, indicating how effectively data flows through the functional test program.

To compute the program's "connectivity", the authors convert the instruction trace of the functional test program into a Control and Data Flow Graph (CDFG). The graph is then traversed, either in the forward or backward direction [5], to follow two different types of operations:

- Read-After-Write (RAW), i.e., instructions that update a destination (i.e., write into a register or a memory location) and then use this value (i.e., read it).
- Write-After-Write (WAW), i.e., instructions that update the value of a destination twice (i.e., write) in sequence without using (i.e., reading) the first one.

Each WAW edge implies an information loss, and the instruction that writes the lost value is marked as useless. On the contrary, each RAW edge correctly propagates a value between registers or memory locations. WAW and RAW edges allow the authors to color the graph nodes as "good" or "bad" and produce high-level metrics representing whether the functional test programs effectively carry their computed value to a diagnostic point, i.e., a Software signature in memory or a Hardware diagnostic register. To provide fine-grain feedback to test engineers, the graph analysis is combined with the executable file and the source code to locate code lines affecting the computed metrics.

The methodology is illustrated in Figure 2.1. The execution trace on the left-hand side runs three instructions that, for the sake of simplicity, write a value in a single destination. The first and second instructions write an immediate value to a register, whereas the last sets the sum of R0 and a value into R1. It is easy to understand that the third instruction immediately overwrites the data written in R1 by the second instruction. For this reason, the second instruction is useless and marked as "bad" (i.e., black). The color of the three nodes is then used to compute the overall connectivity and to rectify the problem in the functional program.

Instruction	SRC	DST
li R0, 1		R0
li R1, 2		R1
add R1, R0, 1	R0	R1

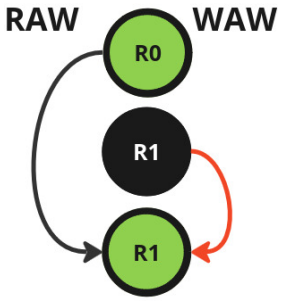

Figure 2.1. RAW and WAW analysis of an assembly code snippet.

Chapter 3

Proposed Methodology

This thesis proposes a strategy to early deploy SLT Functional procedures directly on a specialized ATE architecture [1]. As shown in Figure 3.1, the proposed method exploits the functionalities of an SLT-oriented ATE to validate SLT Functional programs and quickly grade their testing ability based on a connectivity metric extended from [3] to multi-core applications.

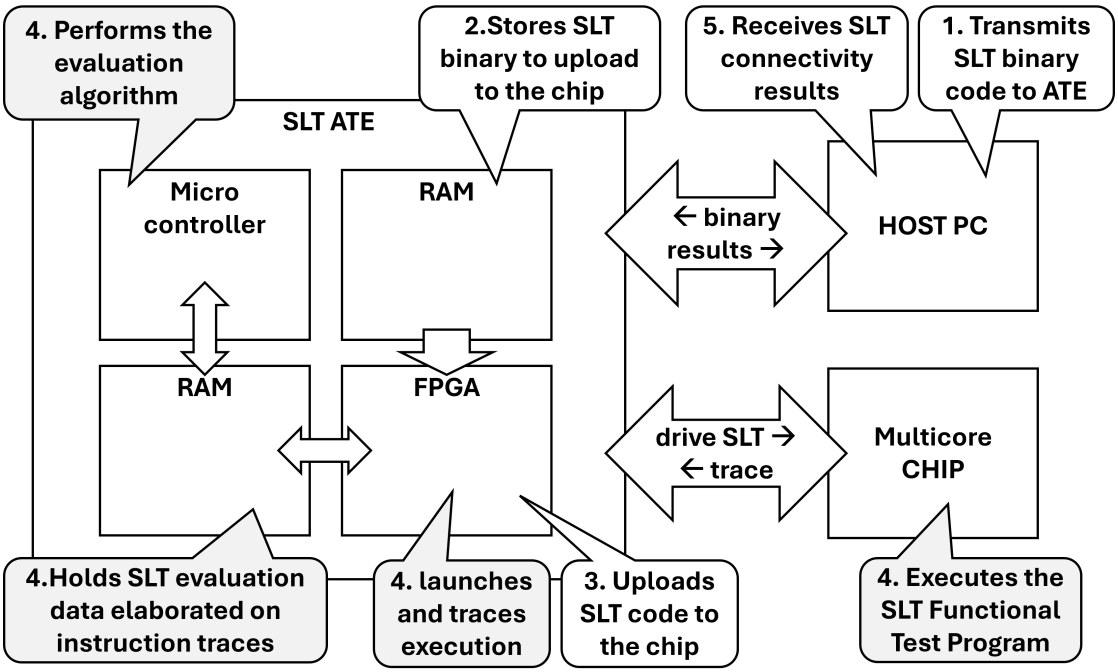

Figure 3.1. Overall view of the proposed approach.

The validation and grading flow in Figure 3.1 starts from the host PC, which is in charge of transmitting the binary code of a SLT program to be applied by the SLT ATE (i.e., a compiled and executable version of the SLT Functional program). The host PC communicates with the computational resources of the ATE, which are usually contained in a microcontroller. This microcontroller coordinates many functionalities related to the application and evaluation of the

SLT. It drives the circuitries mapped on FPGA to upload the SLT functional program, launch it via a functional reset, and trace the SLT program execution by retrieving information about the executed instruction through the chip debug features. Such a trace download is based on triggering a debug trap that permits the extraction of the current instructions' opcode, operands, and memory elements content where needed (for example, for load and store instructions). Trace information, normally stored in a dedicated RAM, is not rawly transmitted to the host PC but elaborated on the fly onboard the ATE, again by the microcontroller resources, as soon as they are available and in parallel to the tracing operations. In such a way, compared to a traditional approach in [3], where quite a large amount of data needs to be transmitted to the host PC, only a small amount of information is returned by the ATE, thus saving time and minimizing the necessity to implement a communication protocol to synchronize the ATE and the host PC. The development of SLT procedures on the ATE triggers several industrial benefits and simplifies the tasks of application engineers responsible for transferring SLT code from simulation/emulation/debugger-based development setups to the ATE ecosystem.

3.1 Multi processor connectivity computation

The connectivity strategy introduced in the background section analyzes each trace instruction by instruction to find the ones that harm the final coverage. Instructions that do not propagate any value are considered useless to improve the final coverage, as every signature computation performed along the program does not depend on them. Furthermore, a value may not be propagated because a subsequent instruction overwrites it or because no subsequent instructions read it.

When an application runs on multiple master cores, such as CPUs, DSPs, DMA controllers, or AI HW Accelerators, each one manipulates its registers and shared memory locations, which can also be used to communicate with the other cores. This means that the destinations written by an instruction on a core can be read by subsequent instructions executed by the same core or instructions running on other cores. For this reason, the connectivity evaluation is extended in this thesis to evaluate RAW edges that span the activity of different cores. Significantly, the proposed approach explained in the following progresses the state-of-the-art [3, 9, 2, 12] by pioneering the evaluation of SLT conceived for multicore computations.

The two code snippets reported in Figure 3.2 illustrate the difference between the single-core and the multi-core algorithms. Destinations are marked in purple and sources in blue for ease of reading. Squared nodes represent load and store instructions, whereas register-based instructions are circle-shaped nodes. Instruction 5 running on core 0 is a load instruction that writes to the memory location 0xF6CA; this location is later read by instruction 6 of core 1. If the algorithm evaluates each trace separately, the instructions are considered "unconnected" and not contributing to fault coverage, as represented on the CDFG on the left-hand side in yellow. On the contrary, if the evaluation considers multi-core behaviors, the instructions working on memory location 0xF6CA become "connected" and can be green-colored as a value is propagated from one core to another. Since that memory destination becomes green, other instructions for the first core become connected, such as instructions 1 and 3 for core 0.

Conversely, the multicore analysis can correctly determine that instruction 3 of core 1 (marked in red), also written to the same memory address 0xF6CA, gets overwritten by core 0.

The order of execution of instructions that modify shared memory is an important factor. Synchronization points implemented in the SLT Software (i.e., semaphores) ensure that a core correctly reads information written by another core. This analysis makes pinpointing dependency arcs between writing and reading operations among multiple cores easy. Accordingly, this

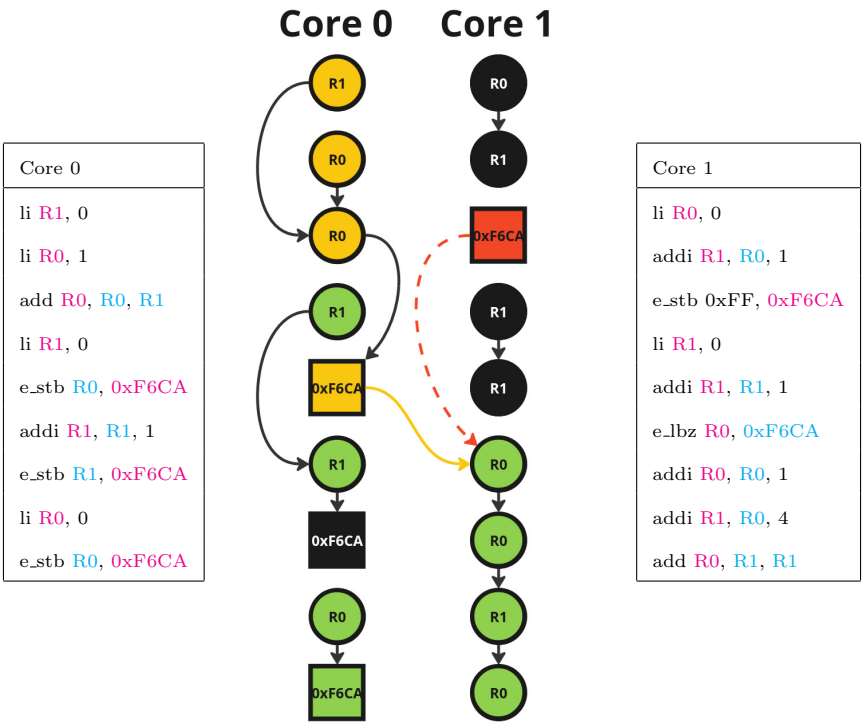

Figure 3.2. Single and multi-core connectivity.

information is exploited to evaluate the correct placement of synchronization points, bringing three pieces of information to the attention of the programmer:

- Uninitialized memory read: the presence of reads at memory locations that were never explicitly initialized. The delay of a write operation could have moved the initialization phase after the corresponding reading phase.
- Missing synchronization between read and write operations: a reading instruction of a data value written by another operation but without synchronization. It represents a read operation delayed long enough to swap its order with a write operation.
- Writes without subsequent reads: the presence of memory writes without an explicit read. This could be caused by a situation similar to the first example.

These three cases represent situations where the code is most likely to be incorrectly synchronized. This could lead to out-of-order executions of the instructions involved and catastrophically impact the SLT testing effectiveness.

3.2 On-tester quick computation of the connectivity

Figure 3.3 illustrates the difference between possible SLT effectiveness evaluation flows. So far, the most used strategy is based on Simulation (SIM) and Fault Simulation (FSIM). This approach increasingly becomes unpractical, as Simulation and Fault Simulation require prohibitive CPU time. Once a SLT suite of programs is produced after SIM/FSIM, an additional step is needed to move it on the tester, called ATE Deploy.

Conversely, indirect measures that leverage the on-chip execution to extract instruction traces largely save time by trading off the accuracy of the estimation. If the ATE architecture is used, a very useful synergy is established among test and application engineers because it minimizes the efforts to deploy the SLT to the ATE.

In Figure 3.3, offline and online possibilities are displayed. The offline version is based on the transfer of full traces from the ATE and assumes an application engineer passes data to crunch to the test engineer. On the contrary, a single test engineer works on the ATE by following the online philosophy, where the support supplied by the application engineer is minimized together with the evaluation time.

The main idea behind the online version of the indirect measurement approach is to interleave the computation of SLT metrics, such as the Connectivity, directly along the execution of the SLT program driven by the ATE on a good chip. As shown in Figure 3.1, FPGA circuitries on the SLT ATE permit the retrieval of instructions' information by implementing traps programmed through the debug features of the chip. Then, the microcontroller of the ATE performs the connectivity computation. The system is optimized by implementing a pipeline-like mechanism visually illustrated in Figure 3.4. The efficiency is maximized if the connectivity computation for instruction i takes the same or less time than the successive instruction $i+1$ trace. In multicore SoCs, consecutive instructions may come from different cores.

The algorithm conceived to compute the connectivity on the fly is illustrated in Figure 3.5. On the left hand, the execution trace of the SLT program presented in Figure 3.2 is shown in the final form. Instructions from cores 0 and 1 are regularly interleaved because the two CPUs run at the same frequency and in a fully parallel fashion. Nevertheless, the order of the instructions from different cores may be irregular. The illustrated algorithm does not care about the core provenance of the instruction and works in both conditions.

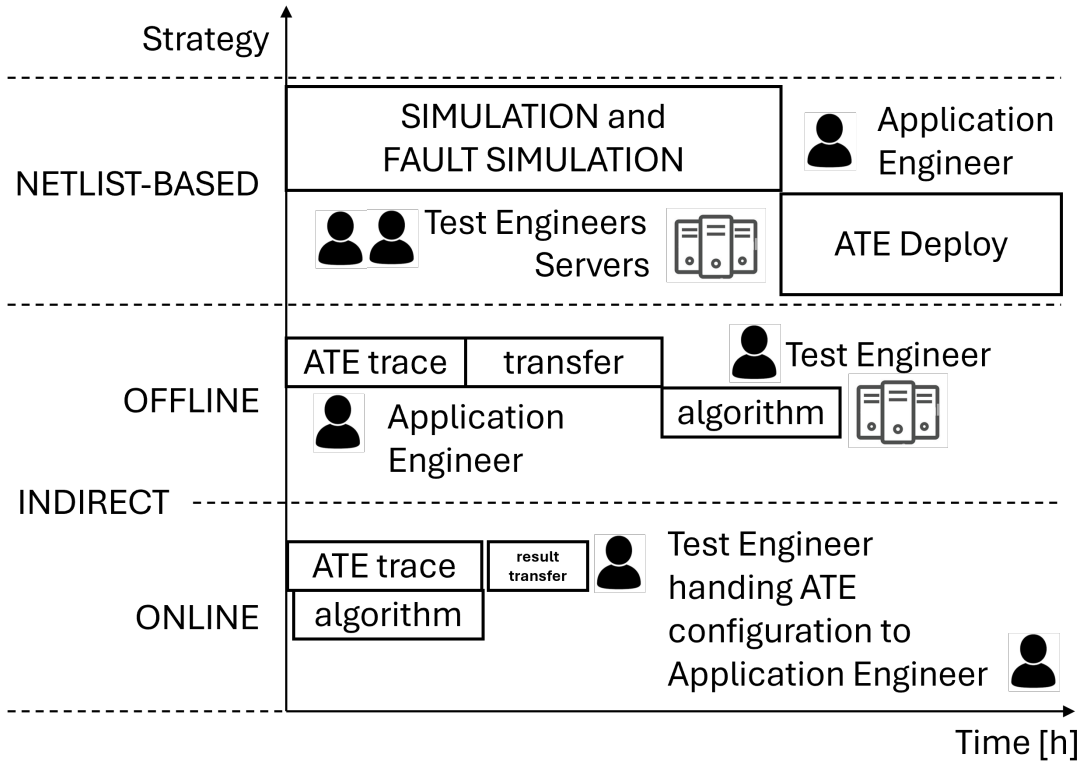

Figure 3.3. SLT effectiveness measurement flows.

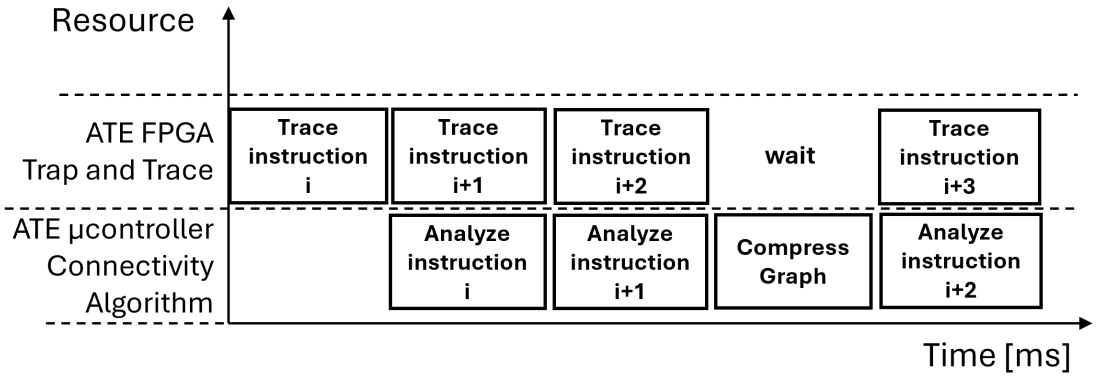

Figure 3.4. Online evaluation of SLT applications.

The graph on the right side of Figure 3.5 results from the elaboration of the multicore trace. Every node represents a data value, which is extracted by the instruction and could be a register value or a memory location. Green nodes represent values readable by future instructions and

contain the current value of the registers or memory location. Black nodes store values that do not propagate to the end of the trace in any way. Finally, gray nodes represent values propagated to the end of the trace but are not readable anymore by any future instructions.

In this new representation, edges are used to keep track of RAW operations defined in the previous work [3]. A node pointing to another means that the pointed node reads the node's value, so information transported by the analyzed instruction is currently propagated. A new, fictitious node called End Of Trace (EOT) is introduced, marking the program's end. Since no further instructions are performed after this node and the content of the core registers are supposed to be checked for potential execution mismatches, any node pointing to the EOT, directly or indirectly, is considered green.

Notice that, as previously specified, the graph is updated every time the debugger generates a new instruction. When a new instruction is produced, the ATE microcontroller performs the online procedure described in Algorithm 1.

Core	Instruction
0	li R1, 0
1	li R0, 0
0	li R0, 1
1	addi R1, R0, 1
0	add R0, R0, R1
1	e_stb 0xFF, 0xF6CA
0	li R1, 0
1	li R1, 0
0	e_stb R0, 0xF6CA
1	addi R1, R1, 1
0	addi R1, R1, 1
1	e_lbz R0, 0xF6CA
0	e_stb R1, 0xF6CA
1	addi R0, R0, 1
0	li R0, 0
1	addi R1, R0, 4
0	e_stb R0, 0xF6CA
1	add R0, R1, R1

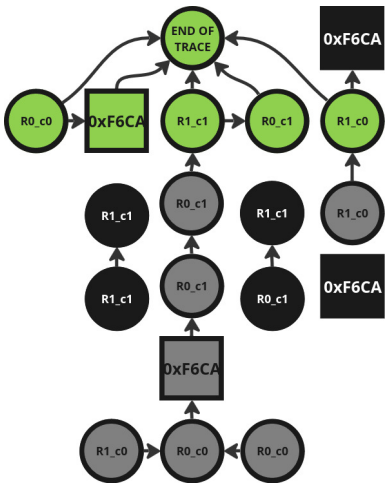

Figure 3.5. Online multi-core connectivity computation: code (left-hand side) and corresponding graph (right-hand side).

The procedure visits all instructions within the trace, or when it works online, it examines each new instruction as soon as this is generated by the debugger (line 1). Then it loops over all of its destinations (line 2), creating a new node for each one (line 3). Looping over the sources of the current instruction, the algorithm connects them to the new nodes (line 7) before detaching any old node sharing the same destination from the EOT node (line 11). The pseudo-code is reported in Algorithm 1.
At the end of this procedure, each node has an implicit color depending on its outgoing edges:

- A node is green if it has an outgoing edge pointing directly to the EOT. These nodes represent the registers and memory locations currently stored in the core.
- A node is gray if it does not point directly to the EOT but exists along a path from the node to the EOT. These nodes represent the values that have been overwritten, but their information is propagated through other nodes.
- A node is black if a path from the node to the EOT node does not exist. These nodes represent the values overwritten without impacting the execution.

The connectivity is computed by dividing the number of nodes not black by the total number of nodes.

Algorithm 1 CDFG: Online construction.

```
1:  for all (istructions ∈ trace) do
2:      for all (dest ∈ instruction) do
3:          create new_node
4:          new_node.dest = dest
5:          for all (sources ∈ instruction) do
6:              if (source ∈ graph and ∃ edge source ⟹ EOT) then
7:                  create edge source ⟹ new_node
8:              end if
9:          end for
10:         if (dest ∈ graph and ∃ edge dest ⟹ EOT) then
11:             remove edge
12:         end if
13:         create edge new_node ⟹ EOT
14:     end for
15: end for
```

From the time and memory complexity point of view, the proposed algorithm generates a graph with as many nodes as the number of trace destinations. Indeed, the procedure adds a new node to the graph for each instruction's destination. To reduce the RAM occupation of the microcontroller, it is possible to merge graph nodes into single nodes or delete useless nodes whenever possible. This optimization is referred to as graph compression in Figure 3.4 and can be implemented by adding a counter for each node data structure. This node information represents the number of destinations the node keeps track of. In general terms, the compression is based on removing nodes already reaching a final "black" state and merging "gray" nodes connected exclusively. According to RAM limitations, the compression step is not necessarily applied after every instruction analysis; it can be performed only time by time. Figure 3.6 shows the CDFG of Figure 3.5 in its memory-optimized version.

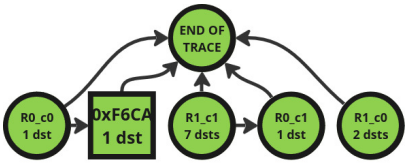

Figure 3.6. Memory-optimized version of the CDFG.

Chapter 4

Experimental results

SLT Functional Program	Executed Instructions	Est. FSIM Time [h]
SingleCore RTOS	108,235	7,154.92
AMP RTOS	145,848	9,641.34
AMP RTOS - fixed	145,829	9,640.09
SMP RTOS	154,493	10,411.14
SMP RTOS - O1	175,227	11,583.45
SMP RTOS - Conserve stack	157,642	10,420.99
SMP RTOS - Address anchor	157,648	10,421.38
SMP RTOS - No omit frame pointer	165,299	10,927.15
XBAR SLT app 1	364,293	24,081,74
XBAR SLT app 2	376,580	24,893,99
XBAR SLT app 3	664,265	43,911.51
XBAR SLT app 4	14,196,439	938,461.54

Table 4.1. Different SLT applications used for the experiments.

Several tests have been run on a 40 nm automotive System-on-Chip of the SPC58 family from STMicroelectronics and reported in this section. The device is a multi-core chip that includes three PowerPC VLE ISA CPUs, which were developed for safety-critical applications in the automotive field. The chip includes approximately 20 million gates, and its three CPUs show around 1.5 million port-level faults.

4.1 Experimental setup

The experimental setup is a development station that includes all elements of a single-site SLT ATE. It addresses the SPC58 micro-controller which is hosted on a socket and connected to a ATE driver board, including a microcontroller and a Xilinx Ultrascale+ MPSoC FPGA, based on a previous work [6]. The setup is shown in Figure 4.1. All programs are loaded on the Device Under Test (DUT) through the FPGA circuitries. FPGA design also cares of trapping the execution after each instruction to extract the information the connectivity algorithm needs. The overall system hosts 4GB of RAM accessible by the MPSoC. The microcontroller system runs on several cores and is controlled by a custom Ubuntu-like version. By running several OS

SLT Functional Program	Connectivity metric [%]			
	Core 0	**Core 1**	**Core 2**	**Multicore**
SingleCore RTOS	NA	NA	72.4	72.4
AMP RTOS	23.71	NA	75.19	76.79
AMP RTOS - fixed	70.15	NA	73.89	74.87
SMP RTOS	79.15	79.15	73.93	75.57
SMP RTOS - O1	82.87	82.87	70.71	72.83
SMP RTOS - Conserve stack	79.15	79.15	73.89	75.53
SMP RTOS - Address anchor	79.15	79.15	73.89	75.53
SMP RTOS - No omit frame pointer	88.36	88.36	76.72	78.08
XBAR SLT app 1	85.15	84.76	90.14	87.74
XBAR SLT app 2	84.33	84.13	89.72	86.97
XBAR SLT app 3	91.88	92.06	91.13	91.65
XBAR SLT app 4	83.21	83.17	83.07	83.15

Table 4.2. Single-core Multi-core connectivity differences.

SLT Functional Program	Offline Execution Time [s]			
	Dump Trace	**Transfer**	**Analysis**	**Total**
SingleCore RTOS	3,720	152	66	**3,938**
AMP RTOS	3,360	208	90	**3,658**
AMP RTOS - fixed	4,680	204	93	**4,977**
SMP RTOS	6,660	219.2	92	**6,971.2**
SMP RTOS - O1	5,040	243.2	99	**5,382.2**
SMP RTOS - Conserve stack	3,120	220.8	96	**3,436.8**
SMP RTOS - Address anchor	3,540	222.4	96	**3,858.4**
SMP RTOS - No omit frame pointer	4,500	233.6	100	**4,833.6**
XBAR SLT app 1	13,440	500	816	**14,756**
XBAR SLT app 2	12,840	516.8	2,051	**15,407.8**
XBAR SLT app 3	22,831	411.2	722	**23,963.7**
XBAR SLT app 4	487,927	8,560	4,218	**500,704.7**

Table 4.3. Offline execution times.

threads on several ARM-A53 cores, the ATE microcontroller manages the algorithm execution and communication with the host PC. Advanced features, such as sending live updates of the connectivity, are also provided to the test engineer for implementing feedback loops to increase the SLT program effectiveness.

The execution trace is produced while each core of the SoC under test is running contemporarily. To correctly align the traces of execution, each trace dump containing the information about the currently executed instruction is complemented by a progressive number functioning as a timestamp. Two instructions with the same timestamp are considered to be executed simultaneously on different cores.

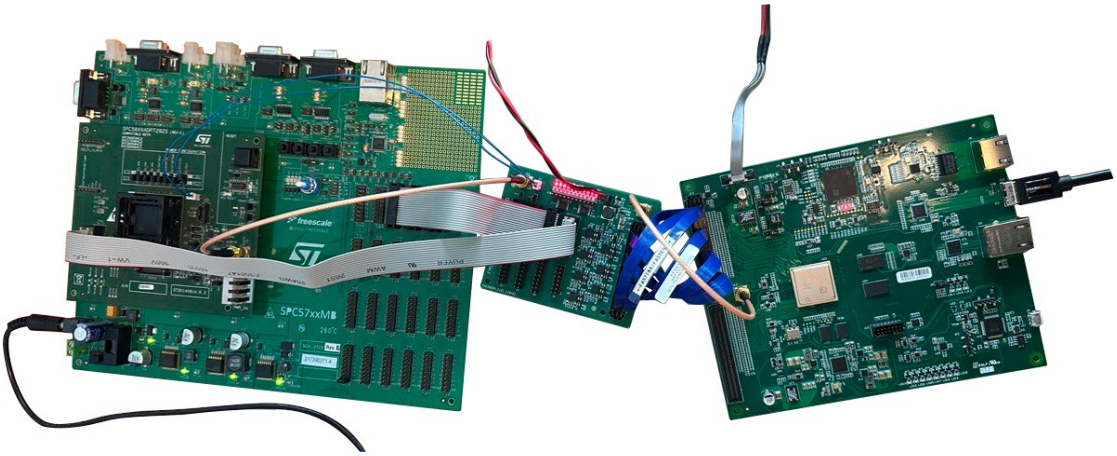

Figure 4.1. Experimental setup with DUT on the left and SLT-ATE on the right.

SLT Functional Program	Online Algorithm Execution Time [s]
SingleCore RTOS	**2,682.1**
AMP RTOS	**2,422.6**
AMP RTOS - fixed	**3,374.3**
SMP RTOS	**4,801.9**
SMP RTOS - O1	**3,633.9**
SMP RTOS - Conserve stack	**2,249.6**
SMP RTOS - Address anchor	**2,552.4**
SMP RTOS - No omit frame pointer	**3,244.5**
XBAR SLT app 1	**9,690.4**
XBAR SLT app 2	**9,257.8**
XBAR SLT app 3	**16,461.2**
XBAR SLT app 4	**351,803.7**

Table 4.4. Online execution times.

4.2 SLT applications Evaluation

The set of SLT Functional Programs listed in Table 4.1 was evaluated using the proposed online indirect approach and compared with the offline version.

The rows of the table are subdivided according to the type of SLT application considered. The table starts with analyzing a Single Core Real-Time Operating System (RTOS), using as starting point $\mu COS - III$ [8], then extended with a custom multicore version. Asymmetric Multi-Processing (AMP) applications exercise two CPUs, one running a bare-metal application, while the second executes a single-core version of the RTOS. At the same time, the Symmetric Multi-Processing (SMP) RTOS treats all three cores equally with a multicore scheduler. Ad-hoc developed SLT applications are reported, too. All programs are written in high-level C language and compiled using optimization level two (basic) as a baseline. For each SLT program, Table 4.1, Table 4.2, Table 4.3, and Table 4.4 report several information.

The column Executed Instructions of Table 4.1 reports the number of low-level assembly instructions retrieved along the trace phase. The SLT suite shows application flows ranging from a hundred thousand to fourteen million traced instructions. Timing-wise, the SLT program execution duration spans from a few hundred milliseconds to some seconds.

Connectivity metrics results for single and multicore SoCs are reported in Table 4.2. The columns named Core 0, Core 1, and Core 2 report the connectivity metric values computed with the single core metric described in [3]; the Multicore column reports the connectivity value measured with the multicore approach proposed in this thesis. Such figures highlight how the single-core connectivity may be significantly far from the most correct one computed over all cores.

The estimated fault simulation CPU time in hours is reported in the FSIM column of Table 4.1. With a medium-sized system like the one used in this work, being able to fault-simulate SLT has become way more utopic than in the past.

Offline and online method performance approaches are finally illustrated in Table 4.3 and Table 4.4. Offline is divided into trace generation, file transfer, and algorithm computation on the host PC. Trace generation results in the most time-consuming activity, with the cost of about 20ms to dump a single instruction. The final column of Table 4.4 is related to the online strategy, and reports its overall execution time, as it is difficult to distinguish between trace generation

and algorithm execution times. On average, the online approach is 33.16% faster than the offline.

The ATE feature enabled a live update from the microcontroller running the algorithm on the host PC, which permitted the extraction of extremely useful insights about the SLT program test effectiveness.

- Figure 4.2 plots the connectivity value evolving along the trace execution. The points on this plot are computed every time a new instruction is added to the graph. Therefore, it can be seen that the connectivity drops from the initial 100% (no instruction evaluated yet) to about 82%. As long as new "good" instructions are evaluated, the connectivity grows, but may again fall, as it happens in the case study. Two minor coverage drops can be identified before a monotonic "slight" drop till the end of the measurement.

- Figure 4.3 shows how many black nodes are discovered for each instruction, along the trace execution. This representation reflects what is happening in Figure 4.2, but it highlights the "badness" of each instruction. In this way, the engineers can identify more easily the instructions that are more problematic.

- Figure 4.4 shows the distribution of black nodes not chronologically, but along the code. The blue plot is the amount of black nodes identified by the instruction at that address, while the orange plot is the number of times the instruction at that address is executed. This graph is even more useful for the SLT developers, who can figure out which instructions of the SLT code are more problematic.

With this live feature, topic instructions or code segments can be identified and used to tune the SLT program and increase its testing quality. In the experimental cases, AMP and SMP RTOS variations span from a basic version to more cured ones, enhanced by tuning compile options or fixing evident deficiencies of the original code emerging from the results analysis, as tables show for XBAR applications that were developed in sequence and addressing each a specific weakness that was highlighted by connectivity drops.

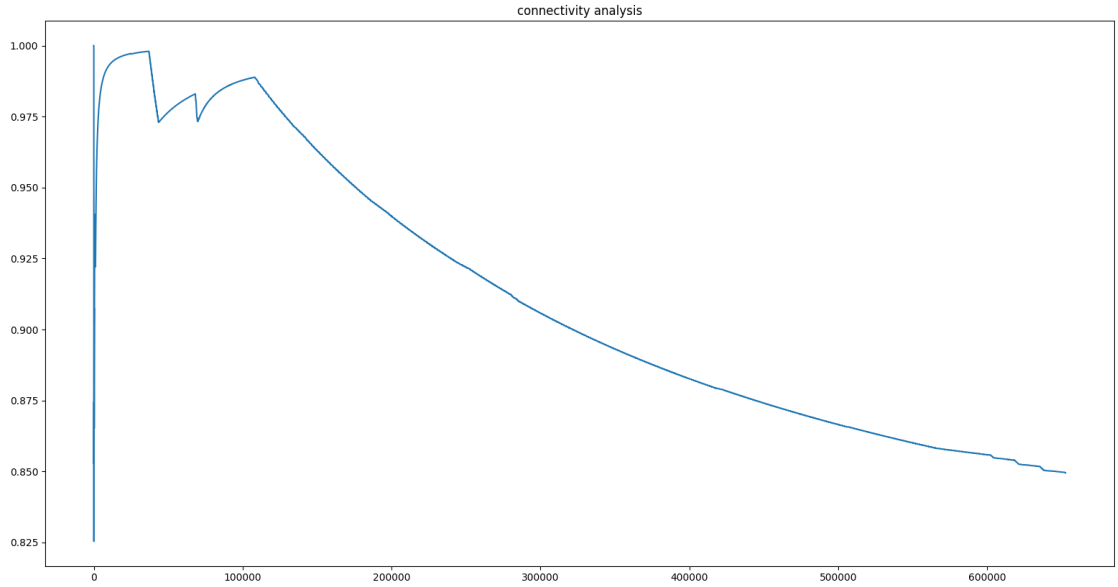

Figure 4.2. Connectivity trend along a SLT functional program execution.

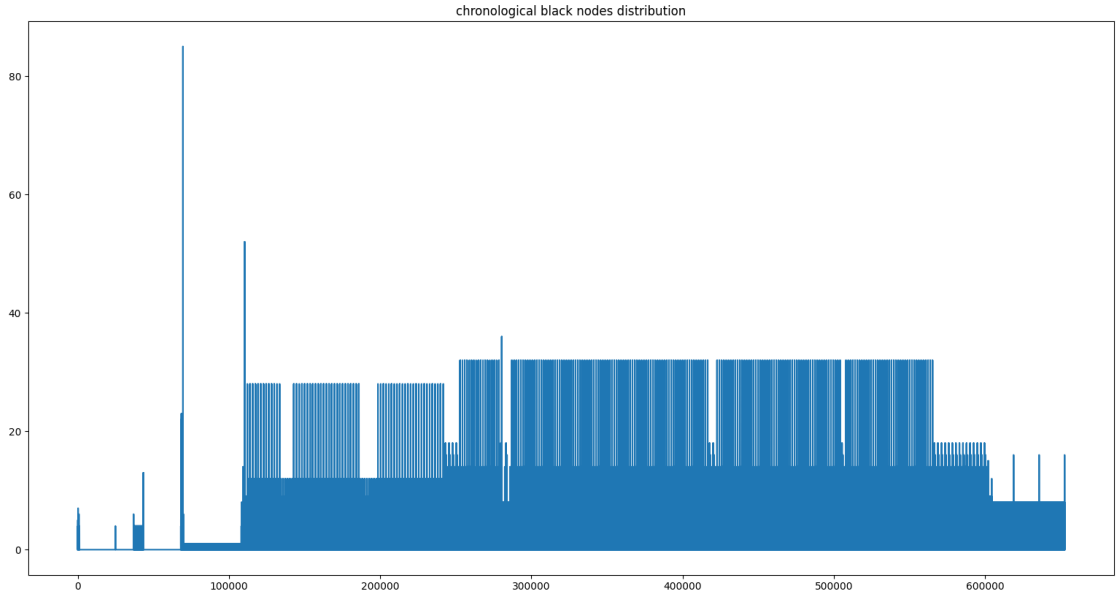

Figure 4.3. Black nodes distributed in chronological order of execution.

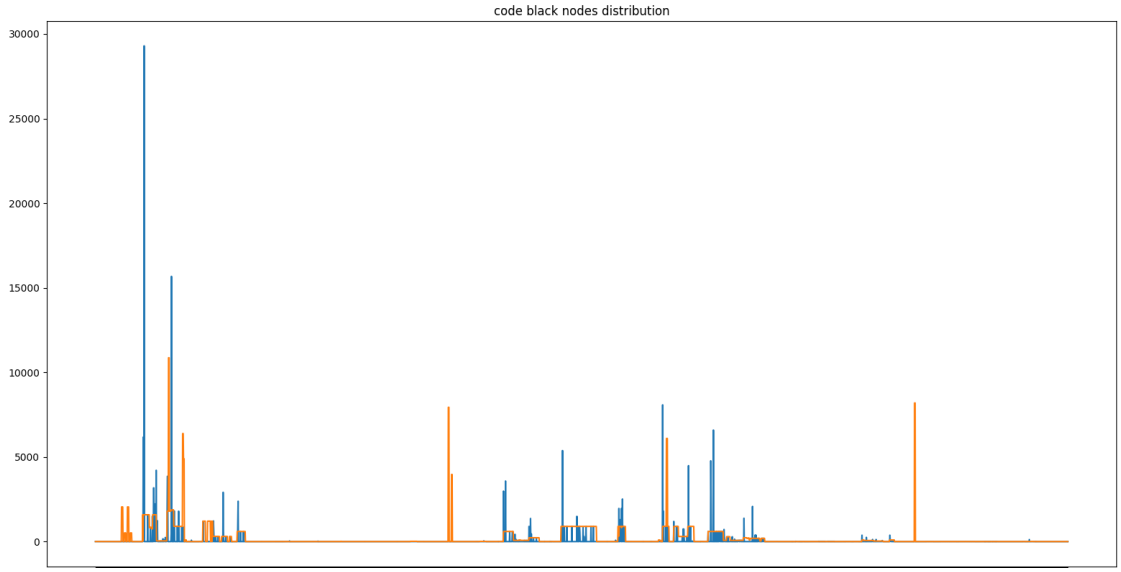

Figure 4.4. Black nodes distribution along the code (blue), with the corresponding flamegraph (orange).

Chapter 5

Conclusions

Starting from this work, there are some future directions that can be followed in order to augment the accuracy of the connectivity evaluation:

- Analysis of the destinations bit-wise: instead of building the Control and Data Flow Graph with destinations as nodes, it is possible to consider every different bit of each register/memory-location as a different node. With this kind of analysis, the accuracy of the evaluation can increase even more, since the data propagation can be inspected in a more precise way. However, the memory complexity of the analysis would increase a lot, because the number of nodes added by a single instruction multiply by the number of bits of its destinations.
- Instruction behavior definitions: in line 7 of Algorithm 1, instead of connecting each reader to each writer, it is possible to create a Read-After-Write edge according to the internal structure of the instruction. Each instruction, depending on which operations it performs, propagates data between its sources and destinations in a different way. It is not true that every destination of the instruction reads always from every source. This kind of improvement requires a deeper analysis of each instruction, which can require a lot of time.

In literature, different indirect methodologies exist to grade SLT applications [3, 12]. This work progresses the state-of-the-art in this topic by proposing an instruction trace-based metric to assess SLT procedures for multicore architectures. The proposed framework is run directly on ATE enabling easy and early handover from test and application engineers.

Acronyms

AI Artificial Intelligence. 11

AMP Asymmetric Multi-Processing. 17, 18, 20, 21

ARM Advanced RISC Machine. 18

ATE Automatic Test Equipment. 1, 5–7, 10, 11, 13, 15, 17–19, 21, 24

CDFG Control and Data Flow Graph. 5, 9, 11, 16, 24

CPU Central Processing Unit. 1, 6, 11, 13, 17, 20

DfT Design for Testability. 8

DMA Direct Memory Access. 11

DSP Digital Signal Processor. 11

DST Destinations. 9

DUT Device Under Test. 5, 17, 19

EOT End Of Trace. 15

FPGA Field Programmable Gate Array. 1, 6, 11, 13, 17

FSIM Fault Simulation. 13, 17, 20

GB Giga-Bytes. 17

HW Hardware. 1, 2, 8, 9, 11

ISA Instruction Set Architecture. 17

MPSoC Multi-Processor System-on-Chip. 17

OS Operating System. 6, 17

PC Personal Computer. 1, 6, 10, 11, 18, 20, 21

RAM Random-Access Memory. 11, 16, 17

RAW Read-After-Write. 5, 9, 11, 15, 24

RTOS Real-Time Operating System. 17, 18, 20, 21

SIM Simulation. 13

SLT System-Level Test. 1, 4–8, 10, 11, 13, 14, 17–22, 24

SMP Symmetric Multi-Processing. 17, 18, 20, 21

SoC System-on-Chip. 1, 6–8, 13, 17, 18, 20

SRC Sources. 9

SW Software. 8, 9, 11

VLE Variable-Length Encoding. 17

WAW Write-After-Write. 5, 9

XBAR Cross-Bar Switch. 17, 18, 20, 21

Bibliography

[1] F. Almeida et al. “Effective Screening of Automotive SoCs by Combining Burn-In and System Level Test”. In: *IEEE DDECS*. 2019.

[2] H. Amrouch et al. “Intelligent Methods for Test and Reliability”. In: *DATE*. 2022. DOI: 10.23919/DATE54114.2022.9774526.

[3] Francesco Angione et al. “An innovative Strategy to Quickly Grade Functional Test Programs”. In: *2022 IEEE International Test Conference (ITC)*. 2022, pp. 355–364. DOI: 10.1109/ITC50671.2022.00044.

[4] Harry H. Chen. “Beyond structural test, the rising need for system-level test”. In: *VLSI-DAT*. 2018.

[5] G. Cabodi et al. “Mixing Forward and Backward Traversals in Guided-Prioritized BDD-Based Verification”. In: *Proc. Computer Aided Verification*. July 2002, pp. 471–484.

[6] Nicola di Gruttola Giardino et al. “A Flexible FPGA-based Test Equipment for Enabling Out-of-Production Manufacturing Test Flow of Digital Systems”. In: *To appear in the proceedings of IEEE DFTS 2024*. 2024.

[7] Naghmeh Karimi et al. “Test generation for clock-domain crossing faults in integrated circuits”. In: *IEEE DATE*. 2012. DOI: 10.1109/DATE.2012.6176505.

[8] Jean J. Labrosse. *UC/OS-III, The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers, DSPs*. Micrium Press, 2009. ISBN: 0982337531.

[9] George Papadimitriou and Dimitris Gizopoulos. “AVGI: Microarchitecture-Driven, Fast and Accurate Vulnerability Assessment”. In: *IEEE HPCA*. 2023. DOI: 10.1109/HPCA56546.2023.10071105.

[10] Ilia Polian et al. “Exploring the Mysteries of System-Level Test”. In: *IEEE ATS*. 2020, pp. 1–6. DOI: 10.1109/ATS49688.2020.9301557.

[11] Denis Schwachhofer et al. “Automating Greybox System-Level Test Generation”. In: *IEEE ETS*. 2023. DOI: 10.1109/ETS56758.2023.10173985.

[12] Denis Schwachhofer et al. “Optimizing System-Level Test Program Generation via Genetic Programming”. In: *IEEE ETS*. 2024, pp. 1–4. DOI: 10.1109/ETS61313.2024.10567817.

[13] Daniel Tille et al. “Towards an Automated Flow for Implementation ofDedicated LBIST Scan Chains for Functional Safety”. In: *TUZ* (2021).

[14] Dilip Kumar Reddy Tipparthi and Karthik Krishna Kumar. “Concurrent system level test (CSLT) methodology for complex system-on-chip”. In: *IEEE EPTC*. 2014. DOI: 10.1109/EPTC.2014.7028421.