On the use of Boundary Scan for Code Coverage of Critical Embedded Software

João Carlos Cunha
Polytechnic Institute of Coimbra/CISUC
Coimbra, Portugal
jcunha@isec.pt

Ricardo Barbosa, Gilberto Rodrigues[1]
Critical Software
Coimbra, Portugal
{rbarbosa, gg-rodrigues}@criticalsoftware.com

***Abstract*— Code coverage tools are becoming increasingly popular as valuable aids in assessing and improving the quality of software structural tests. For some industries, such as aeronautics or space, they are mandatory in order to comply with standards and to help reduce the validation time of the applications. These tools usually rely on code instrumentation, thus introducing important time and memory overheads that may jeopardize its applicability to embedded and real-time systems. This paper explores the use of IEEE 1149.1 (boundary scan) infrastructure and on-chip debugging facilities from embedded processors for collecting the program execution trace during tests, without the introduction of any extra code, and then extracting detailed code coverage analysis and profiling information. We are currently developing an extension to the csXception tool to include such capabilities, in order to study the advantages, difficulties and impediments of using boundary scan for code coverage.**

Keywords*— *software testing; boundary scan; real time; embedded software; code coverage

I. Introduction

Verification & validation (V&V) of business and mission critical software require highly accurate methods, as the effects of failures can go from loss of important mission data or expensive equipment, to loss of human lives. Strictness of testing must be adapted to the severity of failures, meaning that the more severe the consequences of the failure are, the more demanding testing should be, which is only feasible when supported by effective tools. One of the most powerful approaches to test engineers regarding dynamic verification of software is formal coverage criteria.

Code coverage is a measure of systematic software testing, giving the tester a way to decide what kind of tests to use, and to know at which extent the code of the application has been tested. By defining clear requirements for the type and depth of tests, it makes more likely that the tester will find problems in the software, and offers greater assurance of software quality and reliability [1]. This is why some industries are regulated by very strict standards that, among other things, demand testing with full code coverage regarding some specified criteria, for the certification of software. We can find such examples in several standards, such as the RTCA/DO-178B [2] (and the recent RTCA/DO-178C [3]) that regulates the software development, verification and testing for avionics, the ISO 26262 [4] that regulates the functional safety (including software) for the automotive industry, or the IEC 60880 [5], related to software safety in nuclear power plants. To comply with the demands of extensive test coverage criteria, software systems must be tested using automated tools.

Automated testing tools usually rely on the instrumentation of the code, by introducing probes that intercept and control the flow of the application, and collect trace data, without interfering with the sequential behavior of the application. However, this process introduces important time and memory overheads that may put at risk its applicability to embedded and real-time systems. In concurrent applications, the addition of extra instrumentation code causes degradation or simply a change in the system's performance and behavior. For example, some race conditions or other sensitive real-time operations can be masked when running with this extra code. Furthermore, the memory overhead introduced by the tools may limit the execution of tests on the target system's hardware, which usually has limited resources.

A major challenge in embedded systems design consists on the verification of the embedded code while actually executing in the target platform [6]. In fact, the only reason embedded software is commonly validated in testing environments is because of the inexistence of proper tools able to test the target software running in the target hardware, without interfering with its regular operation.

This paper studies the usability of IEEE 1149.1 [7] (boundary scan) infrastructure and on-chip debugging facilities available in most embedded processors to provide automated code coverage analysis of an application running in the final platform, under a simulated environment, without the need for source code alteration or instrumentation. Through the boundary scan JTAG interface, it is possible to transparently access the target system eliminating the toolset footprint, and making execution very realistic, something very important from a safety-critical systems' validation point of view. Under a contract with the European Space Agency, we have developed csXpy, an extension to the csXception tool suite (see Section IV.A) aiming, in a first stage, to explore the viability of such an approach for code coverage of critical space applications.

The remainder of this paper is organized as follows: section II presents some tools and the corresponding technologies that currently assist on software testing through

[1] Gilberto Rodrigues is currently with MTU Aero Engines, Munich, Germany

code coverage analysis; section III focus on the use of IEEE 1149.1 boundary scan infrastructure for tracing the execution of the target application; section IV presents the csXpy tool and section V describes some architectural decisions, implementation details, and preliminary results; section VI concludes the paper.

II. Code Coverage in Embedded Software Testing

Special tools have been proposed, both commercially and at research level, to cope with the difficulties of testing embedded systems and to comply with the required coverage criteria. These tools span from purely software-based to hardware-based, and may be classified according to their operation principle as:

- *Source Code Instrumentation* – with these tools, extra code is inserted at the testing application, at source or bytecode level, in order to collect trace information. This information, stored at the tool's database, is then used to calculate coverage according to several supported criteria. These tools assume that the behavior of the application is not affected by the additional trace calls, which may not be valid for multi-tasking applications, or in limited resources situations (including time). Many of these tools do not support assembly-level instrumentation, which complicates the coverage of low-level interfacing code. Examples of such tools are LDRA Testbed [8], VectorCAST [9] or Cantata [10].
- *Hardware Level Monitoring* – these tools use special hardware to monitor the embedded application in a nonintrusive manner. An example is the Embedded Trace Macrocell [11], an in-chip hardware module, able to collect instruction and data traces, and deliver them off-chip through a high-speed port, or on-chip into an embedded trace buffer. Such tools are usually expensive, and rely on proprietary solutions, restricted to a few target systems. G-Cover [12] is another example, which uses a probe to connect to the target via a more common JTAG/BDM interface.
- *Instruction Level Simulation* – consists of simulating the target CPU at instruction level, which may support code coverage at this level. These simulators are usually slow, and must be qualified before they can be used for credit. For safety-critical applications, these simulators do not provide enough confidence and a real target system is mandated to be used for certification purposes. Moreover, not many simulators are available to cope with some of the most used embedded processors, such as the PowerPC and ARM. An example of such a tool is the TSIM simulator [13], which is an instruction set simulator for the SPARC/ERC32 processor. It is a qualified simulator that can be used for performing validation campaigns within the space industry.

The real appreciation for these tools can be noticed while testing a safety-critical system such as an avionics component: GPS receiver, flight control system, or any other graded as DAL-A/B as per RTCA/DO-178B/C. These standards demand that all requirements-based tests must be executed on the target system for coverage purposes. In this situation, several constraints are presented by the system, that prevent the use of typical testing methodologies to achieve the necessary credit for certification.

For example, in the case of source code instrumentation, the representativeness of the executing application is diminished by the extra calls needed to provide coverage (instrumentation of the target source code). This implies that the tests are required to run at least twice: one for executing the tests without the coverage features for obtaining the testing results, and a second one with instrumentation for coverage purposes, in order to compare results and assure the instrumentation code did not alter its behavior. Although usually accepted as good enough, this is not a solution for non-deterministic applications, such as concurrent programs. Using special purpose hardware overcomes these disadvantages, but suffers from the enormous cost and risk of developing hardware, which even requires specific validation activities. Finally, the use of instruction level simulators is not even considered in this industry, since developing a certified simulator representative of the real target would cost the same (or even more!) as developing the actual application.

One possible solution to overcome the problems of interfering with the target system due to software instrumentation or to monitoring embedded hardware, is to use the already available infrastructure introduced and certified within these target systems – JTAG – usually present in the most common (and specific) embedded target platforms. A tool using this technology can easily be qualified and, with the track record it already presents, is relatively inexpensive to implement.

III. Using Boundary Scan for Test Coverage

As a way of guaranteeing that all the components in a printed circuit board are correctly connected, and to cope with the difficulty (or impossibility) of inserting physical probes, circuit manufacturers designed shift register test cells connected to every I/O pin of each component, which are able to control, read and write the state of the I/O pins, allowing the test of the board interconnections. The standard that describes this architecture is IEEE 1149.1, commonly known as JTAG [7]. These test cells are disposed in a chain of boundary scan cells, as depicted in Fig. 1.

The data contained in the whole scan chain cells can be shifted out through the Test Data Out (TDO) port, while new data is simultaneously shifted in through the Test Data In (TDI) port. A Test Access Port (TAP) controller is able to control the instants the boundary scan cells collect the functional data from the device, or output data into the device cells.

In modern processors, each component – Integer Unit (IU), Floating-Point Unit (FPU), Memory Controller (MEC), On-Chip Debugger (OCD), etc. – has its own chain of boundary scan cells. In normal circuitry functioning, these boundary scan cells are completely transparent having no

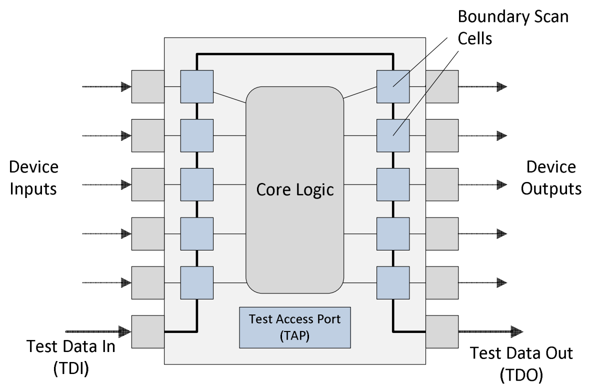


Figure 1. Chain of boundary scan cells

influence in the integrated circuit operation. However in test mode they are able to read and write any pin or register cell from the device.

Since the OCD registers are accessible through the JTAG interface, this feature is becoming broadly used to debug software that runs in embedded systems. This JTAG-based debugging uses the OCD to insert breakpoints, execute in single-steps and, by accessing the IU, FPU, MEC and the CPU pins, can read and write registers and even memory, by way of controlling the read/write cycles through address, data and control pins and buses [14].

The access to these boundary scan chains provides the necessary means to trace the application execution at object-level, without performing any change in the application software. It is only necessary to introduce hardware breakpoints in code or data memory addresses and execute in single-step by accessing the OCD registers through boundary scan. When the CPU freezes, it is then possible to collect all sort of information through the proper boundary scan chains such as, for example, the currently executing address, a general purpose register, a memory cell or even a pipeline register.

The principles used by all the debuggers that rely on boundary scan and on-chip debugging capabilities are basically the same: they introduce breakpoints in memory addresses and collect the system state through boundary scan. However, while debuggers only sporadically need to freeze the CPU for collecting data, code coverage tools need to trace the application execution, thus must collect system state continuously.

Some architectures already offer these trace capabilities, such as ARM (Embedded Trace Macrocell [11]), MPC56x or MPC55xx (using NEXUS [15]), or MPC4xx (RISCWatch [16]). However, in order to be able to collect trace data in real-time, high-speed ports must be accessed by expensive trace modules, such as those from Lauterbach (JTAG based disassemblers and debuggers [17]). ARM also offers a solution with a low-speed JTAG port, by storing trace data in an in-chip circular Embedded Trace Buffer [18]. The drawback of this solution is that the amount of trace information collected by the host system is, of course, limited by the different rates data is inserted and retrieved from this buffer.

IV. A Boundary Scan Code Coverage Tool

csXpy was initially developed under a contract with the European Space Agency (ESA) for the creation of a monitoring and profiling tool for space software. The goal was to design and develop a non-intrusive code coverage and profile analysis tool for the SPARC/ERC32 processor, used by ESA, able to perform automated code coverage analysis over the original non-instrumented code.

The aim of csXpy was thus to be able to conduct such test experiments and provide extensive code coverage analysis, including Statement Coverage (SC), Decision Coverage (DC), Branch Condition Coverage (BCC), Entry/Exit point Coverage (EEC) and Modified Condition Decision Coverage (MC/DC), as required by RTCA/DO-178B/C standards for software certification regarding safety and reliability in an airborne environment.

Beyond code coverage, the tool would also be able to provide accurate timing information of the running application, both at task level (e.g. task CPU time) and operating system executive level (e.g. context switch).

A. The csXception Framework

csXception [19] is a fault injection environment developed by Critical Software [20], used in the evaluation of critical systems. csXception performs state-of-art product certification on the dependability/RAMS area, by providing an environment for performing fault injection based tests. csXception was originally built as a Software Implemented Fault Injection (SWIFI) tool, able to emulate the effects of hardware faults, by means of software exception handling modules embedded in the target system, based in different architectures, such as Intel x86, PPC or Sparc. This tool has been evolving into a tool suite, with added modules such as the Easy Fault Definition (EFD), a user interface providing the tester facilities to define faults in an easier way, based, among others, on the visualization of the target functional modules (e.g. registers), and the software with corresponding memory map. Another module, the Xtract (Xception Analysis Tool) allows the tester to easily analyze the results, for example, the conditions of the system in the moment a fault was injected or detected.

Besides SWIFI, csXception is also able to inject software faults, both in C and Ada languages source code, and to take advantage of the boundary scan chains of target systems to perform Scan Chain Implemented Fault Injection (SCIFI).

A fault injection campaign begins by defining, through the EFD running in a host system, all the parameters for fault injection, such as the fault triggers, the fault model (type, location and duration), and the workloads to execute during fault injection. Typically some hundreds or thousands of faults are defined, and then the campaigns run automatically.

Faults are injected in a target system either through some software modules embedded in the target (as in SWIFI) or through a boundary scan controller (as in SCIFI). A host system (where faults were defined) communicates with the target for controlling the experiments and collecting all data in a database.

Regarding SCIFI, the host starts by setting up the hardware breakpoint conditions (through boundary scan), and then the workload begins its execution until it is halted by reaching a breakpoint. Then the host injects the fault by reading the contents of the scan chains, changing the bits according to the defined fault data, and writing back through shifting in the scan chains into the target. The workload is resumed and the experiment follows the general proceedings. At the end of each injection run the tester has updated information about the ongoing campaign.

B. *csXpy Requirements*

Considering the SCIFI features already available in the csXception framework, it has been decided to take advantage of these features to build csXpy and study the applicability of boundary scan to test coverage. In fact, csXception already provides many features necessary to csXpy, such as: an environment for defining the parameters for the experiments; a database for storage of experiments parameters and results; a communication environment with the target system through a boundary scan controller; a tool for analyzing the results.

Nevertheless, several changes had to be made, namely: the definition of test cases is definitely different from the definition of faults: the process of injecting faults requires the programming of a single breakpoint (usually a single fault is injected), while for code coverage multiple breakpoints need to be (re)defined (more on this in the next section); the processing of the results in code coverage is also different from fault injection, although some facilities provided by Xtract, such as source code visualization, may be used.

The main requirements for the csXpy code coverage and profiling tool were basically the following:

- To achieve code coverage assessment on-target, without the need for additional instrumentation. At least MC/DC (Modified Condition Decision Coverage) was required;
- To provide precise timing information about the execution of the test application.

C. *How csXpy Works*

In order to get test coverage information at source code level, it is necessary to have a detailed mapping between each piece of source instructions and the corresponding assembly instructions. So, the first step of csXpy is to parse the source code of the target application, written in C language, in order to identify every statement, decision, condition, function entry and exit point. Then these tokens are matched with the binary instructions and the corresponding memory addresses. This is performed with the help of javaCC [21] parser, and the debug information from gcc [22] compiler. This information is stored in csXpy database for coverage analysis at the end of each test run.

Furthermore, every branch instruction detected at assembly level (jumps, calls, return, etc.) is also signaled during this phase, as well as the target address. This is also stored in csXpy database, but for controlling the execution of the tests aiming the collection of the execution trace log.

Fig. 2 presents an example of the decomposition of a C-language *if()* instruction into assembly-level instructions for a SPARC architecture. This *if()* instruction corresponds to a decision, composed by several conditions (boolean expressions, without logical operators). It is thus possible to make the correspondence between each condition in the source code and the branch instruction that tests it at object level.

If no source code exists, csXpy still collects addresses of the detected branch instructions, and thus is able to provide coverage information at assembly level.

The next phase consists of monitoring the execution of the tests using JTAG and the OCD facilities of the target processor. The application binary code is uploaded into the target system, either remotely or from a local ROM, and its execution is traced by introducing breakpoints at branch instructions and trap table, thus catching every change on the control flow. At these points the CPU is frozen, including the internal clock, and csXpy collects the state of the processor, namely the program counter and a timestamp. In brief, the following steps are performed from the moment the embedded application is ready to start executing:

1. At the beginning of a test-run, a hardware breakpoint for program execution is set at a user-defined starting-point address (usually defined at source-level, and mapped to a memory address). Another breakpoint for data memory is set for the entire trap table address space, for catching any software exception or external interrupt.
   The application then runs until a breakpoint condition is reached.

(a) C source instructions

```
if (((d != a)&&(b==2))||(c < a))
    c=a+b;
```

(b) Assembly code

```
 ...
 0x2001274 <Init+64>: cmp %o0, %o1
 0x2001278 <Init+68>: be 0x20012a0 <Init+108>         ; (d != a)
 ...
 0x200128c <Init+88>: cmp %o1, 2
 0x2001290 <Init+92>: be 0x20012c4 <Init+144>         ; (b == 2)
 0x2001294 <Init+96>: nop
 0x2001298 <Init+100>: b 0x20012a0 <Init+108>
 0x200129c <Init+104>: nop
 0x20012a0 <Init+108>: ...
 ...
 0x20012b0 <Init+124>: cmp %o1, %o0
 0x20012b4 <Init+128>: bl 0x20012c4 <Init+144>        ; (c < a)
 0x20012b8 <Init+132>: nop
 0x20012bc <Init+136>: b 0x20012e4 <Init+176>
 0x20012c0 <Init+140>: nop
 0x20012c4 <Init+144>: ...                            ; true block
 ...
 0x20012e4 <Init+176>: ...                            ; end if
```

Figure 2. Example of (a) C-language i*f()* statement decomposition into (b) assembly-language instructions

2. The next branch instruction that shall be executed is then detected, either by analyzing the csXpy database, or by disassembling the code sequence. The hardware breakpoint register is then set with this branch instruction's address.
3. The CPU is restarted, running at full speed until a programmed breakpoint is reached.
   If a trap occurs (due to an exception or external interrupt), this event is caught by the breakpoint that is monitoring the accesses to the trap table.
4. This breakpoint or trap position is logged by csXpy, as well as an internal timestamp. Then, the CPU runs in single-step until the destination of the branch is reached (which is also logged, in order to have information about a taken/not taken conditional branch). This information is collected from the CPU status registers, accessed through the corresponding boundary scan chains.
5. The process is repeated from step 2.

When the application ends (either due to a timeout or reaching a predefined condition) the collected information is able to be analyzed, off-line, and presented to the tester. By following the log it is possible to track the complete execution trace of the application and, by comparing the executed instructions and destination of branches with the parsed information already stored in csXpy database, it is possible to detect if a statement has been executed, if a condition has been evaluated as TRUE and/or FALSE, or if a decision has been evaluated as TRUE and/or FALSE. This allows the calculation of code coverage information, such as:

- Statement Coverage – percentage of statements that were reached during execution. This corresponds to reaching the nodes of the application control flow graph (the nodes are basic blocks of statements, i.e., sequences of statements that always execute sequentially).
- Decision Coverage – percentage of decisions that were evaluated as both TRUE and FALSE. This corresponds to following the edges of the control flow graph.
- Branch Condition Coverage – percentage of conditions evaluated as both TRUE and FALSE. A decision is composed of one or more conditions.
- Entry/Exit Point Coverage – percentage of points of entry and exit (e.g. function calls) that have been invoked at least once.
- MC/DC (Modified Condition Decision Coverage) – percentage of conditions that determined the logical result of the decision, i.e., that have independently affected the decision's outcome.

D. *csXpy Setup*

As already mentioned, csXpy takes advantage of its integration into the csXcepton framework. It consists of a control application running as a plugin of csXception host, which is connected to the target (embedded) system through a JTAG controller, as presented in Fig. 3.

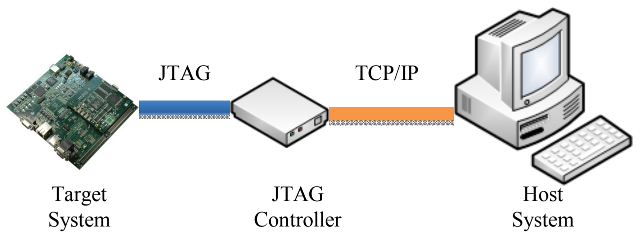


Figure 3. csXpy external view. The csXpy software runs only in the host system, and controls the target (embedded) system through JTAG.

The JTAG controller hides from the host system the particularities of JTAG operations. It is able to fully control the target system through the boundary scan chains test access ports (TAP). This controller receives from the host typical debugging commands, such as "initialize", "read register", "set hardware breakpoint", "single-step", etc. and accesses the OCD and other boundary scan chains to execute them. We have used a commercial JTAG hardware debugger from VisionMC [23].

csXpy currently supports the SPARC/ERC32 architecture, in a TSC695F board [24].

E. *csXpy Test Suite Execution Cycle*

The execution of a test suite, i.e., a set of test cases, with the csXpy tool, follows basically the general steps described below:

1. A test suite is created by defining some parameters, such as the identification of the target system and target application, through the csXpy graphical user interface plugin running in the csXception host system.
2. Then the tester has to specify the test cases, consisting in the definition of the workload, test inputs and expected outputs.
3. The tests in the test suite run automatically. For each test case, the target system is initialized, binary code is uploaded (or is automatically copied from ROM), initial hardware breakpoints are set, and the target starts execution. While running, trace data is collected by csXpy in the csXception database. Remember that these operations are controlled by csXpy running at the host system: it keeps sending commands and receiving data through the JTAG controller.
4. At the end of each test run, trace data stored in the csXception database is analyzed, and metrics and performance are calculated.
5. While the desired coverage levels are not reached, further tests may be created by the tester and executed.

Throughout all this process, the tester has always available updated information in the csXpy user interface, such as:

- The current coverage metrics (Statement Coverage, Decision Coverage, etc.), per test and for the whole test suite (see Fig. 4.a);
- Timing results (Fig. 4.b). The user defines pairs of locations in code (e.g. from start to exit of a directive) and obtain results about minimum, maximum, average time and variance between consecutive accesses to these locations;
- A visual view of code coverage results, at source code level (Fig. 4.c). For each statement, condition or decision, and for all coverage criteria, a color code indicates if it has been covered, or not covered. It is possible to “enter” in each statement, e.g. an *if()* instruction, to get detailed information, such as regarding each condition.

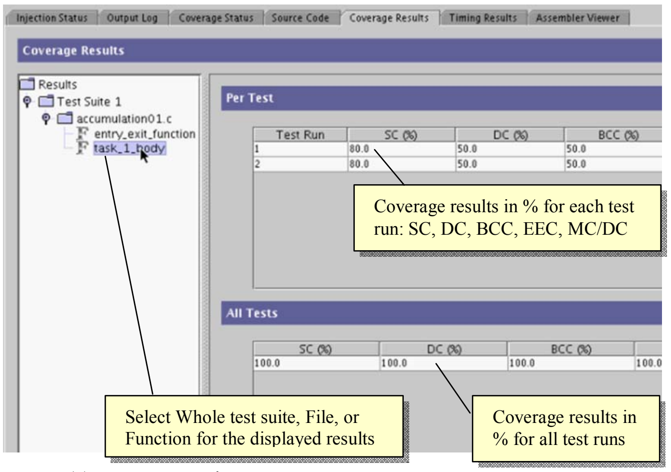
This allows the tester to continuously monitor tests progress, and to stop when the intended coverage level is reached.

F. *Coverage Results*

As already mentioned, it is through the analysis of the trace execution log against the mapping between the source code and assembly that the necessary information is extracted to, for example, classify statements as executed, or decisions as covered.

Taking as an example another *if()* instruction written in C language for RTEMS [25] in the ERC32 architecture, its equivalent assembly code, which is given by *sparc-rtems-objdump*, is represented in Fig. 5. While executing the application at the target system, csXpy generates a trace containing the addresses of all control flow changes (such as branch instructions) and the next address, to permit the calculation of branch taken or not taken. This information allows drawing conclusions about the decision statement in the source code. In this example, the trace log indicates that after the address *2001868*, the application continued to address *200186c*, meaning that branch instruction *be 200188c* was not taken, which results in the conclusion that the condition *res!=RTEMS_SUCCESSFUL* was TRUE in this execution.

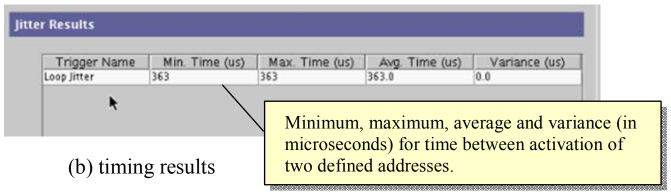
Continuing this analysis throughout the entire log, the tool provides the tester detailed information about how thoroughly the code was executed. The accumulation of the



Test Run	SC (%)	DC (%)	BCC (%)
1	80.0	50.0	50.0
2	80.0	50.0	50.0

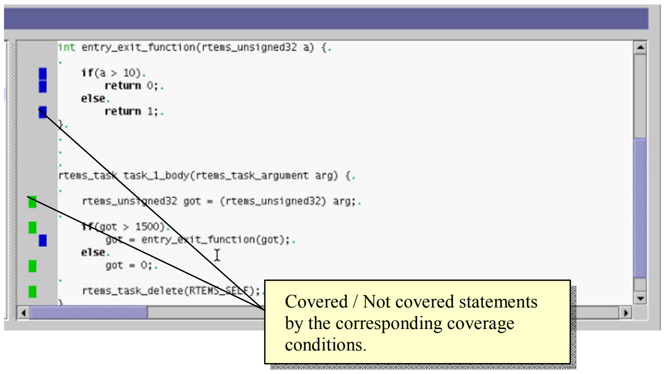
SC (%)	DC (%)	BCC (%)	
100.0	100.0	100.0	100.0

(a) coverage results


Jitter Results
Minimum, maximum, average and variance (in microseconds) for time between activation of two defined addresses.

Trigger Name	Min. Time (us)	Max. Time (us)	Avg. Time (us)	Variance (us)
Loop Jitter	363	363	363.0	0.0

(b) timing results


Covered / Not covered statements by the corresponding coverage conditions.

```
int entry_exit_function(rtems_unsigned32 a) {

   if(a > 10)
      return 0;
   else
      return 1;
}



rtems_task task_1_body(rtems_task_argument arg) {

   rtems_unsigned32 got = (rtems_unsigned32) arg;

   if(got > 1500)
      got = entry_exit_function(got);
   else
      got = 0;

   rtems_task_delete(RTEMS_SELF);
}
```

(c) source code coverage visualization

Figure 4. GUI view of csXpy test results: (a) code coverage, (b) timing information, (c) source code coverage visualization.

(a) C source

```
if (res != RTEMS_SUCCESSFUL) {
    printf("Could not create
  mutex\n");
    exit(-1);
  }
```

(b) Assembly code

```
if (res != RTEMS_SUCCESSFUL) {
2001860:   d0 07 bf e8    ld  [ %fp + -24 ], %o0
2001864:   80 a2 20 00    cmp  %o0, 0
2001868:   02 80 00 09    be  200188c <Init+0x64>
200186c:   01 00 00 00    nop
     printf("Could not create mutex\n");
2001870:   13 00 80 53    sethi  %hi(0x2014c00), %o1
2001874:   90 12 63 80    or  %o1, 0x380, %o0 ! 2014f80
<__DTOR_END__>
2001878:   40 00 35 8a    call  200eea0 <printf>
200187c:   01 00 00 00    nop
     exit(-1);
2001880:   90 10 3f ff    mov  -1, %o0  ! ffffffff <RAM_END+0xfdbfffff>
2001884:   40 00 33 d0    call  200e7c4 <exit>
2001888:   01 00 00 00    nop
  }
```

(c) Trace Log

```
2001868
200186c
```

Figure 5. Example of (a) C source code and (b) the corresponding assembly instructions. (c) the trace log stores the address of each branch and the address of the next instruction

results from a set of tests permits to calculate coverage metrics for each criterion. For example, to calculate Decision Coverage for the test suite, it is necessary to divide the number of decisions from the application that have been evaluated at least once in any test to both TRUE and FALSE, by the total number of decisions in the application.

It is worth explaining how MC/DC coverage is calculated. Basically it is necessary to count the number of conditions that have determined the result of the decision, and divide by the total number of conditions. A condition is said to have determined the result of the decision if, in the log, there is at least one pair of situations where the condition has been evaluated to both TRUE and FALSE and the decision has been evaluated to both TRUE and FALSE (not necessarily in the same order), while the remaining conditions of the decision have not been evaluated differently.

V. Design Issues

While designing the csXpy tool we faced some challenges and impairments, leading us to take some options described in this section.

A. *Optimized Compilation*

Code coverage information is obtained by relating the assembly-level branches with the source code control structures (such as decisions, conditions, function calls or returns). This is done during a static code analysis.

At compilation optimization levels other than –O0 (zero optimization), the compiler is able to reorder instructions, unroll cycles, eliminate branches, etc., making it impossible to have a one-to-one relation between the source code conditions and the assembly-level branches, or between source and assembly blocks of sequential instructions. Even for non-optimized compilation, conditions may be eliminated from the assembly, for example if they are never reached in a sequence of conditions inside a decision.

When such a matching is not possible, no coverage information can be generated. In this case the user is only informed of this by highlighting the unmatched conditions.

Since optimization is limited by the industry standards (e.g. DO-178B/C or IEC 60880), due to indeterminism and unpredictability, requirements on code optimization for the csXpy tool were conferred lower importance, as this study was mainly conducted having in mind the validation activities of safety critical applications (avionics, nuclear, etc.).

B. *Time Information*

Every time the CPU is frozen, the internal timers also stop, introducing a zero delay from the viewpoint of the application.

Besides the branch-related information, csXpy also collects a timestamp, with a resolution in CPU clock cycles. With this information it is possible to have precise time information about the moment each instruction has been executed. Depending on the Operating System Kernel, csXpy may also be able to collect information regarding the currently executing task, allowing deducing, for example, about the moment a task switch occurs.

With these possibilities the tester is allowed to define addresses where he wants timing information to be collected, and then obtain the minimum, maximum, and average time and variance between two activation points, which may be, for example, a context-switch time or the time between periodic task activations.

Even for optimized code it is possible to collect precise timing information, since it is only necessary to define two memory addresses (instruction or data). The tester just has to make sure that, if he wants to get time information related to the execution of any high-level instruction, he chooses a valid memory address of a corresponding low-level instruction. The graphical user interface of csXpy assists the tester on this operation.

However there is an obvious situation where the collection of accurate timing information is impossible: when the target system needs to interact with real environment. Timing information is exact from the application and computing system perspective, as the reference timer (the CPU clock) stops when JTAG operations are being executed. However external systems may have different time references, such as the real world clock. This situation is analyzed in the following section.

C. *Environment Emulation*

The non-intrusiveness of the proposed method is limited to only the software components, but is not non-intrusive in the sense of a deployed system that needs to interact with a real environment. In fact, if the application controls or communicates with an external system, its functioning may be affected by the need to freeze the CPU during the JTAG scan process (this is also true for software instrumentation based code coverage tools).

Testing of safety-critical systems is typically made in a thoroughly controlled environment. The test environment is well known and all environmental data is also known, logged and analyzed in detail. Even unprocessed data is recorded for post-testing analysis. The environment is thus simulated by software, and fully controlled, on a different machine.

Moreover, considering a coverage monitoring tool, the requirements for processing external inputs are limited. The main goal is to get coverage of an application running in the target system (final platform), without the need for source code alteration or instrumentation. Until now we have tested the tool only for some sensitive parts of the system under test, not concerning inputs and outputs.

However, even the simulation of environmental scenarios doesn't come without complexity. Since timers stop when the target system's CPU is frozen, the simulation machine must also stop execution, maintaining both timers synchronized. We are planning to do this by using a dedicated line between both machines, as successfully achieved in MAFALDA-RT [26].

D. *Time Overhead*

As already explained, execution trace information is collected every time a branch instruction is executed. It is easy to deduce that stopping execution at each branch instruction is certainly very time-consuming. Tests have revealed that this overhead can go from 800 to 1200 milliseconds in our target system using the commercial JTAG interface. This includes multiple accesses to several CPU scan chains, for collecting and reordering bits into proper register information, single-stepping, programming the OCD hardware breakpoint for next branch instruction, and resuming execution. Moreover, each of these commands is generated by csXpy running in the host system, so it is also necessary to account for the communications between the host system and the boundary scan controller, over a TCP/IP connection.

This large overhead has two main reasons.

First, the access to the scan chains themselves, either for reading the CPU registers, for programming the OCD, or for resuming execution, take a considerable amount of time. In our target system with ERC32 CPU, the JTAG interface operates at a frequency of 8 MHZ, meaning that it takes 125 nanoseconds to read each bit of the scan chains. For a 4751-bit scan chain (the longest scan chain, from the Integer Unit, that must be accessed for collecting CPU registers), this corresponds to almost 600 microseconds just to access this single chain. Since it is necessary to read and write the scan chains several times, it is expected that, at least, some milliseconds will be necessary.

But there is another and most significant reason for the high overhead. The commercial JTAG controller used by csXpy was appropriately developed for debugging purposes, but is extremely inefficient for our purpose of collecting a complete trace of the executing application. In fact, this type of controllers collect several times the scan chains, and rely on several commands from the host system, through the TCP/IP interface (see Fig.3), to execute each debug operation (read/write a register or application variable, read memory, single-step, etc.). This amount of time is perfectly acceptable when the controller is invoked once in a while (for debugging), but very penalizing for collecting such a trace.

The solution to this problem consists thus in eliminating the unnecessary actions, such as the frequent TCP/IP communications or the access to unused scan chains, and in having a more efficient algorithm to log trace data. This will be explored in the next section.

E. Hardware Optimization

Due to some limitations from the commercial JTAG controller, namely the just mentioned performance penalty, a special JTAG controller was built. This controller, named csXbox, was developed in VHDL and deployed in a Xilinx Virtex5 FPGA, consisting of a JTAG controller developed according to the IEEE1149.1 standard, trace buffers, and some glue logic to integrate these modules with a LEON3[27] soft-core. This CPU allowed the installation of Snapgear Embedded Linux [28], and TCP/IP drivers for the communication with the host.

csXbox has achieved a dramatic performance improvement mainly in three ways. First, the csXpy core algorithm was improved, avoiding some simple-stepping and scan chains accesses in every breakpoint. Then, this software was transferred from the csXpy host into the csXbox that, along with a trace buffer, avoids communication between these two modules during test execution. Finally, this hardware has the ability to reorder bits of the boundary scan chains instantaneously, allowing the collection of the intended registers in a faster way. It is worth mentioning that the bits from each CPU register or pins inside the scan chains are totally disordered, reflecting their physical location inside the chip and not the logical order.

With this specially designed board, the csXpy overhead for each branch is now less than 10 milliseconds, meaning that it has been reduced about 100 times.

Some further optimizations are being studied, such as the reduction of breakpoints due to branch instructions. For example, in a loop, if we detect that the next branch instruction is always the same (the branch at the end of a block is being taken to the beginning of the same block), we can then program the next breakpoint to the branch-not-taken destination, thus avoiding several breaks. This and other optimizations are being studied.

F. Retargeting the Solution

Another obvious limitation of this tool is its dependency on the target system. In fact, since it accesses the boundary scan chains, this dependency is total, as each CPU implementation has its own scan chains.

Having this in mind, both hardware and software from csXbox were designed in a way to be easily retargeted to other platforms.

The software is very modular, and only those modules that depend on the target architecture (that deal with processor registers, on-chip debugger, etc.) need to be modified. Furthermore, an API has been specifically designed for developers to create such target specific functions.

On the hardware side, csXbox was also designed in a way that VHDL doesn't need to be changed. For this purpose, a RAM memory has been included in the VHDL, designed to store a lookup table with the identification of each bit in the scan chains, i.e., this table has one entry for each bit in the scan chains (see Fig.6). Thus, each cell of this table contains the identification of the register and order inside the register for each bit from the scan chains. The contents of this lookup memory must be loaded at startup from configuration files. While the scan chain is being shifted out, another RAM memory, acting as a Register File, is being filled in one bit at a time. This Register File is thus dual-ported: from one side it is 1-bit wide, being addressed by the values contained in the lookup table; from the other side it is 32-bit wide, each location corresponding to a 32-bit register. At the maximum it may contain 2048 32-bit registers. The lookup table has 64K entries, allowing scan chains with up to a total of 64K bits.

So, when csXbox starts execution, it must be initialized with target-specific software, which includes both the csXbox core software modules and the boundary scan chains configuration tables.

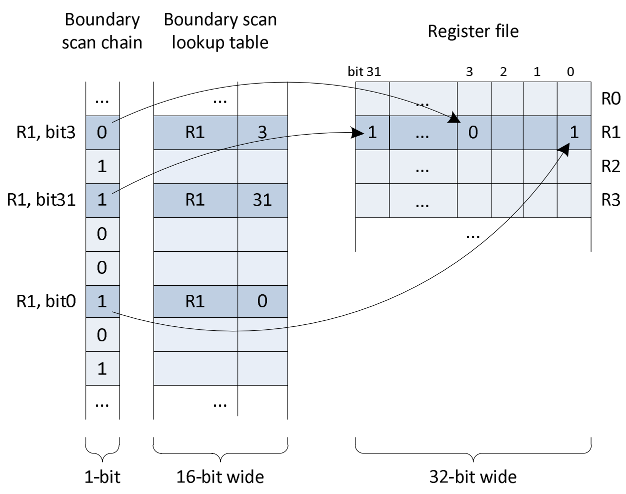


Figure 6. The Boundary scan lookup table provides the location of each bit from the scan chains in the register file.

From the host side, the csXpy plugin running in csXception will need minimum adaptations, providing the source code is still compiled with gcc. For other source languages, more extensive adaptation will naturally be required.

VI. Conclusions and Future Work

This paper analyzes the advantages and problems of using boundary scan for code coverage. A tool, csXpy, was built for this purpose, being able to non-intrusively monitor software execution at runtime, log the execution trace and, from this log, provide detailed code coverage information.

It achieves this by parsing an application written in C to create a mapping between branches, decisions, function entries etc. and the corresponding assembly instructions, and storing that information in a database. It then uses the JTAG and on-chip debugger facilities of the target processor to monitor execution of the application. Using the stored database, breakpoints are set at each next branch instruction and the processor is allowed to run at full speed until the next breakpoint. The reached branches (breakpoints) are logged as well as destination of the branches.

In this way, the tool is able to provide code coverage analysis from the logged data. Furthermore, the tool also logs a timestamp on reaching each breakpoint. Therefore, it can also provide timing information of an application, besides code coverage analysis.

The paper also describes a custom built hardware, csXbox, which addresses some limitations of commercial JTAG debugging tools. It contains a JTAG controller with added hardware to access the boundary scan chains, control the application execution in a faster way and store the trace log in an internal memory. This special hardware decreased the overhead of stopping execution at branch conditions, by about 100 times.

From the experience gained in building csXpy we can draw the following conclusions related to the use of boundary scan for code coverage:

- This approach offers a fundamental advantage over tools that instrument the applications, or need special on-chip hardware, as the target software and hardware do not need to be adapted for testing.
- Precise timing information can be obtained with the same resolution of CPU clock.
- Due to the fact that the CPU freezes during boundary scan chain accesses, it is impossible to test applications that operate with external systems, which have their own time reference. It may operate, though, in simulated environments, for which we are currently exploring synchronization issues with the environment simulation system.
- Only non-optimized programs can be tested, as mapping between high level language source code conditions and assembly language branches is impossible for optimized code.
- The overhead due to the access to the boundary scan chains every time a branch instruction is reached can be huge. On an ERC32 based system with JTAG interface operating at 8 MHZ, the IU scan chain takes almost 600 microseconds to be shifted out.
- A commercial JTAG controller may reveal highly inefficient for tracing the execution of the application due to multiple accesses to the scan chains and the heavy communication with the host for getting all the necessary commands. A solution might be a special-designed JTAG controller, which runs the tracing algorithm and avoids communication with the host during each test run.
- Even with this optimization, the overhead related to a single branch in the object code is of almost 10 milliseconds. Experiences revealed that a simple RTEMS system call such as *rtems_semaphore_obtain()* takes 19 microseconds to run at full speed, and takes almost 500 milliseconds to run with csXpy. This means that this technique will probably not be used to test full embedded applications, but only specific parts of the system, such as system calls, the scheduler, or the dispatcher. For this purpose, it is highly advantageously, as it introduces no time or memory overhead.

We are planning to enhance the tool to be able to support other target platforms, such as LEON2, and other source-level languages, such as Ada. We are also planning to integrate the tool in development environments, such as Eclipse.

Although the use of boundary scan for code coverage doesn't seem to be suitable for testing a full embedded application, it may be of extreme utility for testing specific parts of the system, such as system calls, scheduler, or dispatcher. Since no modifications are made to the software code, csXpy provides a unique tool for test coverage and profiling of sensitive parts of safety-critical systems.

ACKNOWLEDGMENTS

This paper was partially supported by the European Space Agency – European Space Research and Technology Centre (ESA-ESTEC)

REFERENCES

[1] P. Ammann and J. Offutt, "Introduction to Software Testing", Cambridge University Press, 2008.

[2] "DO-178B, Software Considerations in Airborne Systems and equipment Certification", RTCA, Prepared by SC-167, December 1992.

[3] "DO-178C, Software Considerations in Airborne Systems and Equipment Certification", RTCA, Prepared by SC-205, December 2011.

[4] "Road vehicles – Functional Safety – Part 6: Product Development: Software Level", ISO 26262, December 2009.

[5] "Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions", IEC 60880, May 2006.

[6] G. Karsai, F. Massacci, L. J. Osterweil, and I. Schieferdecker, "Evolving embedded systems", in IEEE Computer, May 2010, pp.34-40.

[7] "Standard Test Access Port and Boundary-Scan Architecture", IEEE Std. 1149.1, 1990.

[8] "LDRA Testbed". Internet: www.ldra.com, visited on September 13th 2012.

[9] "VectorCAST". Internet: www.vectorcast.com, visited on September 13th 2012.

[10] "Cantata". Internet: www.qa-systems.com/cantata.html, visited on September 13th 2012.

[11] "Embedded Trace Macrocell™ Architecture Specification", ARM, 2009

[12] "Safety Critical Products: G-Cover Object Code Analyzer", Green Hills Software Inc. Internet: www.ghs.com/products/safety_critical/gcover.html, visited on September 13th 2012.

[13] "TSIM ERC32/LEON Simulator", Aeroflex Gaisler. Internet:www.gaisler.com/cms/index.php?option=com_content&task=view&id=38&Itemid=56, visited on September 13th 2012.

[14] L. E. Frenzel, "The Embedded Plan For JTAG Boundary Scan", Electronica Design. Internet: electronicdesign.com/Articles/ArticleID/19626/19626.html, visited on September 13th 2012.

[15] "NEXUS 5001 Forum™". Internet: www.nexus5001.org, visited on September 13th 2012.

[16] "RISCWatch Debugger", IBM. Internet: www-01.ibm.com/chips/techlib/techlib.nsf/products/RISCWatch_Debugger, visited on September 13th 2012.

[17] "Lauterbach® Development Tools". Internet: www.lauterbach.com, visited on September 13th 2012.

[18] "ARM®, Embedded Trace Buffer Technical Reference Manual". Internet: infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0242b/index.html, visited on September 13th 2012.

[19] "csXception Automated Fault-Injection Environment". Internet: www.csxception.com, visited on September 13th 2012.

[20] Critical Software website. Internet: www.criticalsoftware.com, visited on September 13th 2012.

[21] "Java Compiler Compiler (JavaCC) - The Java Parser Generator". Internet: javacc.java.net/ visited on September 13th 2012.

[22] "GCC, the GNU Compiler Collection". Internet: gcc.gnu.org, visited on September 13th 2012.

[23] "ERC32 MDS", Vision Microsystems. Internet: www.visionmc.com/english/products/erc32/erc322.html, visited on September 13th 2012.

[24] "Evaluation Board TSC695 User Guide", Atmel Corporation, 2005.

[25] "RTEMS driver documentation", Gaisler Research, February 2009.

[26] M. Rodríguez, A. Albinet, and J. Arlat, "MAFALDA-RT: a tool for dependability assessment of real-time systems", in Proc. Int. Conf. on Dependable Systems and Networks, Washington, D.C., June 2002.

[27] "Leon 3 Processor", Gaisler Research. Internet: www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53, visited on September 13th 2012.

[28] "Linux for LEON processors", Gaisler Research. Internet: www.gaisler.com/cms/index.php?option=com_content&task=view&id=63&Itemid=31, visited on September 13th 2012.