J Electron Test (2012) 28:469–486
DOI 10.1007/s10836-012-5316-1

Time-Constraint-Aware Optimization of Assertions in Embedded Software

Viacheslav Izosimov · Giuseppe Di Guglielmo · Michele Lora · Graziano Pravadelli · Franco Fummi · Zebo Peng · Masahiro Fujita

Received: 6 November 2011 / Accepted: 27 June 2012 / Published online: 18 July 2012

Abstract Technology shrinking and sensitization have led to more and more transient faults in embedded systems. Transient faults are intermittent and non-predictable faults caused by external events, such as energetic particles striking the circuits. These faults do not cause permanent damages, but may affect the running applications. One way to ensure the correct execution of these embedded applications is to keep debugging and testing even after shipping of the systems, complemented with recovery/restart options. In this context, the executable assertions that have been widely used in the development process for design validation can be deployed again in the final product. In this way, the application will use the assertion to monitor itself under the actual execution and will not allow erroneous out-of-the-specification behavior to manifest themselves. This kind of software-level fault tolerance may represent a viable solution to the problem of developing commercial off-the-shelf embedded systems with dependability requirements. But software-level fault tolerance comes at a computational cost, which may affect time-constrained applications. Thus, the executable assertions shall be introduced at the best possible points in the application code, in order to satisfy timing constraints, and to maximize the error detection efficiency. We present an approach for optimization of executable assertion placement in time-constrained embedded applications for the detection of transient faults. In this work, assertions have different characteristics such as tightness, i.e., error coverage, and performance degradation. Taking into account these properties, we have developed an optimization methodology, which identifies candidate locations for assertions and selects a set of optimal assertions with the highest tightness at the lowest performance degradation. The set of selected assertions is guaranteed to respect the real-time deadlines of the embedded application. Experimental results have shown the effectiveness of the proposed approach, which provides the designer with a flexible infrastructure for the analysis of time-constrained embedded applications and transient-fault-oriented executable assertions.

Responsible Editor: C. Metra

V. Izosimov
Embedded Intelligent Solutions (EIS) by Semcon AB,
Linköping, Sweden
e-mail: viacheslav.izosimov@eis.semcon.com

G. Di Guglielmo (✉) · M. Lora · G. Pravadelli · F. Fummi
Dept. of Computer Science, University of Verona,
Verona, Italy
e-mail: giuseppe.diguglielmo@univr.it

M. Lora
e-mail: michele.lora@univr.it

G. Pravadelli
e-mail: graziano.pravadelli@univr.it

F. Fummi
e-mail: franco.fummi@univr.it

Z. Peng
Dept. of Computer and Information Science,
Linkoping University,
Linköping, Sweden
e-mail: zebo.peng@liu.se

M. Fujita
VDEC, University of Tokyo, CREST – Japan Science and Technology Agency,
Tokyo, Japan
e-mail: fujita@cad.t.u-tokyo.ac.jp

Keywords Fault-detection optimization · Software-level fault tolerance · Time-constrained embedded software · Transient fault · Soft error · Executable assertion

1 Introduction

In the recent years, the exponential increase of digital-circuit performance has been followed by an increasing requirement of system dependability. Indeed, the continuous reduction of transistor sizes and threshold voltages makes the circuits more sensitive to spurious voltage and charge variations that may result in *transient faults* [7]. Transient faults are intermittent and non-predictable faults caused by external events, such as energetic particles striking the chip. These faults do not cause permanent damages, but may affect the running applications by altering data and execution flows. When a transient fault alters the application execution, such a phenomenon is defined as a *soft error* [45].

Several approaches have been developed for hardening safety- and mission-critical systems against transient faults and soft errors [32]. In particular, massive use of redundant hardware components has been deployed to satisfy high dependability requirements in the domains that are very susceptible to electro-magnetic phenomena, such as nuclear, medical, and space domains. However, in addition to high-cost of the redundant-hardware solutions, the duplication of components introduces significant speed, power, and area penalties, making them unacceptable in most of the designs. Moreover, designers of high-end-system applications should benefit from commercial off-the-shelf (COTS) components for performance and time-to-market reasons. Indeed, COTS components are typically based on newer technology solutions than the hardened ones, i.e., commercial components are more powerful and less power demanding. This is due to the longer design cycles which the hardened components require before obtaining the dependability certification [6], [52].

In such a context, the requirement of guaranteeing dependability over unreliable-hardware components has moved the research focus to the software part of the systems. Techniques for detecting, and possibly correcting, the effects of transient faults, i.e., soft errors, have been referred as software-level fault tolerance (SLFT) [21]. In Fig. 1, the hardware-level fault tolerance (HLFT) approach is compared with the SLFT approach. The failures, i.e., the user-visible effects of soft errors caused by transient faults, may range from minor inconveniences, e.g., having to restart a smartphone, to catastrophic events, e.g., real-time-software crash in an aircraft that prevents the pilot from recovering. On the left side of the figure, HW-shielding and HW-hardening approaches aim at preventing, detecting, and correcting transient faults at electrical, gate, and architectural levels. On the other side, SLFT approaches do not require

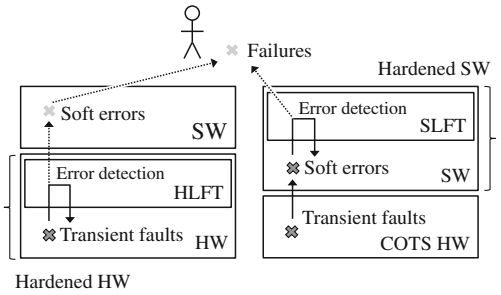

Fig. 1 Hardware- vs. software-level fault tolerance

any hardware modification. Moreover, they can provide different protections against soft errors for individual applications or portion of applications to achieve various dependability requirements. If a hardware-level solution is used, the same protection mechanism will be used for the different application, while different software-level protection mechanism can be used for the different applications to satisfy the different dependability requirements. Finally, SLFT approaches may complement other already existing error-detection mechanisms in off-the-shelf components, e.g., error-correcting code (ECC) on memories and integrity-checks mechanisms on I/O peripherals.

In literature, several SLFT approaches have been proven to be effective in addressing soft errors that alter both data and execution flow of applications [21]. An overview of the state of the art is reported in Section 2. In the present work, we restrict our analysis to the optimization of soft-error detection via *executable assertions* in the context of time-constrained embedded applications.

Independently from the adopted solutions, the hardened software in existing SLFT approaches both performs the originally-specified functionalities and executes additional special operations, e.g. assertions, to detect soft errors. The execution of such error detection operations do not affect general applications, but may potentially compromise timing of *real-time applications*. In a real-time application, the correctness of an operation depends not only upon its behavioral correctness, but also upon the time in which it is performed: missing a time constraint (or deadline) is a total failure for *hard real-time* applications, while, in *soft-real time* applications the usefulness of a result degrades after its deadline. In literature, there are examples of time-constrained applications which may be compromised by soft-errors both in high-end and low-end market: medical-surgical-nursing applications [28], anti-lock brake systems, community storage solutions [33], e-commerce transactions [9], etc. All of these examples may benefit from error-detection approach implemented at software level, provided that the time constraints are not affected.

Thus, in this work, we assess timing efficiency of assertions in order to reduce performance overheads of error

detection and propose a framework for selecting, among the whole set of assertions that could be placed into time-constrained embedded software, a subset of them with, possibly, the highest error detection capability and the lowest execution overhead. The proposed framework will:

1. identify candidate locations for assertions;
2. associate a candidate assertion to each location (the candidate assertion is suggested by the framework, but the user can change it);
3. statically/dynamically profile the module with assertions (inspired by the work of Hiller et al. [23]); and
4. select a sub-set of assertions in terms of performance degradation and tightness (by using the optimization infrastructure) to guarantee an effective error detection capability without negatively affecting the deadline of the targeted embedded SW applications.

The presented approach is useful for optimization of executable assertions in embedded applications with timing constraints. Examples of such systems include, but are not limited to, automotive electronics, airborne software, factory automation, and medical and telecommunication equipment. The main advantage of our approach is that it can be automatically applied at the source-code level, without the intervention of the designer, e.g., for selecting where to put the executable assertions and manually profiling them. Moreover, the approach is completely independent on the underlying hardware, and it may complement other existing error-detection mechanisms.

The remaining of the paper is organized as follows. Section 2 presents the works in literature addressing hardware- and software-level error-detection approaches and, in particular, it focuses on their limitations in the context of time-constrained embedded applications. Section 3 presents the adopted application model, describes principles of error detection, and discusses basic properties of executable assertion. Section 4 outlines the problem formulation and describes the proposed assertion-placement optimization framework. Section 5 reports experimental results. Finally, Section 6 is devoted to concluding remarks.

2 Related Work

Transient faults and soft errors in digital systems can cause abnormal behaviors and failures that compromise system dependability. This is especially true in nuclear, medical, and space environments, where many transient-fault causes, including alpha-particles [27] and ray neutrons [19], are highly likely and a system failure can have catastrophic outcomes [28], [33], [9]. Moreover, with the technology shrinking and sensitization, soft errors have become a problem also in non-critical environments and at sea level, e.g., due to radiation impurities in the device materials [63], [8], [10]. In this context, fault avoidance techniques for high-budget and safety-critical domains have been traditionally based on shielding and hardening techniques. In particular, error detection and correction mechanisms through hardware hardening can be introduced at different architectural levels, for example by using massive module redundancy [60], [3], watchdog processors [4], strengthened latch/flip-flops [38], die monitoring [50] or other techniques [42], [32].

On the other hand, commercial-off-the-shelf (COTS) components are typically not hardened against transient faults. In most of the cases, shielding of these components is impracticable either, since different type of shields must be used against different type of particles [31]. At the same time, most COTS components have a higher density, faster clock rate, lower power consumption and significantly lower price. Thus, software-level fault tolerance has become an attractive solution for developing COTS-based dependable embedded systems, since it requires no extra hardware modification and only minor software modifications [58]. In particular, software-level-fault-tolerance (SLFT) techniques aim at detecting, and possibly correcting, soft errors before they manifest as failures, by adding to the embedded applications extra functionalities for error detection.

Since soft errors may affect the running applications by altering execution or data flows, SLFT approaches have been classified accordingly [21]. In particular, *execution-flow-hardening* approaches aim at detecting errors that directly modify the sequence of executed operations in applications, e.g., by affecting branch conditions or assembly-code-fetch operations; while *data-hardening* approaches aim at detecting errors that modify the application results and, indirectly, the execution flow, i.e., the sequence of executed operations. In turn data-hardening approaches make use of (a) *computation duplication* at different application levels, (b) *data-structure protection*, and (c) *executable assertions*.

Hardening the application-execution flow requires to monitor the sequence of operations and, in particular, the control points. The basic idea of control-flow checking [59] is to partition the application program in basic blocks, i.e., branch-free parts of code. For each block a deterministic signature is computed and faults can be detected by comparing the run-time signature with a pre-computed one [2], [37], [55], [20]. The tool presented in [49], i.e., SWIFT, increases dependability by inserting redundant code to compute duplicate versions of all register values and inserting validation instructions before control flow and memory operations. In general,

the SLFT approaches based on execution-flow checking are effective, but error affecting the data may go undetected, i.e., the data errors may not affect the execution flow. Moreover, a problem is to tune the methodology granularity that should be used, that is managing the extra-computation overhead.

The next main category of SLFT approaches concerns hardening of data. The simplest approaches for data hardening are based on computation duplication [43] and may apply at different application levels, i.e., assembly instruction [36], [54], high-level instruction [46], [47], instruction block, function [34], and program [48], [44], [51]. Each of these approaches transforms the code by replicating computation and checking the (intermediate) results for consistency, e.g., the outputs of replicated programs are compared to detect possible errors. In the case of inconsistency, a soft error is detected and possibly an appropriate error recovery procedure has to be executed. The designer can introduce a further granularity level by replicating only the most critical portions of the application [11]. A major limitation of these approaches, when running on conventional processors, is the high performance overhead [13]: the replicas execute in sequence and then the results are compared, i.e., *time redundancy*. Thus, recently, methods have been proposed for exploiting instruction-level parallelism of COTS superscalar processors [36] and multi-core CPUs [48], [44], [51], [26]. For example, EDDI [36] duplicates instructions at compile-time and uses instruction pipelining of superscalar processors. The possibility of executing the same application in different threads [48] or programs [44], [51] is known as *spatial redundancy*: it permits to reduce the performance overhead thanks to some COTS-hardware support. Both time and spatial redundancies are difficult to adapt to soft-error detection in the context of time-constrained embedded applications. Indeed, time-redundancy approaches may introduce excessive time overheads, thus violating application deadlines; while super-scalar and multicore CPUs used for spatial redundancy can be expensive.

The SLFT approaches based on data diversity are similar to ones based on computation duplication: two copy of the same program is executed, but on slightly different data [16], [35]. For example, ED^{4}I [35] compiles the original program in a new one by introducing a "diversity factor k" on integer operations: a soft error is detected if the outputs of the original and transformed program do not differ of the factor k. With respect to traditional computation-duplication approaches, data-diversity approaches involve extra computations to produce the perturbed version of the program and to compare the results of the different runs. Thus, they hardly adapt to time-constrained embedded applications.

Data-structure protection [41] and algorithm-based fault tolerance (ABFT) [24] are other approaches to provide fault detection through data redundancy. They are usually implemented at the application level. ABFT techniques have been widely used for matrix-based signal processing applications such as matrix multiplication, inversion, LU decomposition and Fast Fourier Transforms [26]. ABFT approaches are effective but need a thorough understanding of the algorithms under study. Moreover, they are suited only for specific applications using regular structures, and therefore its applicability is valid for a limited set of problems.

Finally, let us consider executable assertions. In software verification, designers widely use executable assertions [22] for specifying invariant relationships that apply to some states of a computation, e.g., pre- and post-conditions of a procedural-code block. In this case, executable assertions address design errors. Indeed, several assertion-related works target debuggability and testability of applications. For example, C-Patrol [61] automatically introduces executable assertions into C programs. Assertions that are placed at each statement in a program can automatically monitor the internal status of the application. However, the advantages of universal assertions come at a cost: a program with such extensive internal instrumentation is slower than the same program without the instrumentation; moreover, some of the assertions may be redundant. Thus, some authors introduce assertions based on sensitivity properties of the code, i.e., which parts of the code are the most difficult to test and are the most critical for execution of safety-critical applications [56]. Besides design errors, executable assertions can potentially detect also soft errors in internal data caused by transient faults [62]. For example, in [55] and [20], the authors introduce assertions at compile-time to monitor effects of soft errors on data (and control) flows. In [22], the authors propose a methodology to sort out faults from incoming signals with assertions, which can be used to stop propagation of errors caused by transient faults through the system. This work has been later extended with assertion optimization to increase system dependability with profiling in [23]. In [40], an "out-of-norm" assertion methodology describes how to insert assertions into electronic components, in particular, communication controllers, to detect transient faults. Finally, the methodology proposed in [5] integrates assertions into embedded programs with automatic program transformations to provide error detection and recovery against transient faults. Assertions significantly contribute for defining SLFT approaches, but they are not transparent to the designer and, thus, their effectiveness in detecting soft errors largely depends on the nature of the application and on the ability of the designer in optimizing the assertion choice and placement.

To our knowledge, none of the previous works are focused on the problems derived from the overhead introduced by executable assertions in the context of time-constrained embedded software. This paper is intended to fill in the gap by proposing an optimization framework that allows evaluating the impact of assertion execution and select, among the defined assertions, a sub-set with high error-detection capability and low overhead.

3 Executable Assertions for Soft-Error Detection

This section introduces some preliminary concepts. In particular, it describes the adopted application model, the executable assertions for soft-error detection, and the parameter criteria associated with assertion optimization. Using the example of an embedded-application module, we present assertion properties and discuss a methodology for assertion selection.

3.1 A. Application Model

To represent the embedded-application at instruction level, we adopt a *control* and *data-flow graph* structure with conditional edges [15]. In particular, given an embedded application M, a control and data flow graph $G = \{V, E_S, E_C\}$ is associated, where V is a set of nodes (program instructions) and E_S and E_C are sets of simple and conditional edges. Edges can be either data or control dependencies between the nodes. We do *not* require that the graph has to be *polar*, i.e., several source and sink nodes are possible, in particular, with relation to data initialization and parallel program executions. In our model, a node $I_i \in V$ is a module instruction and an edge e_{ij} is a direct dependency between instructions I_i and I_j, which can be either data or logical dependency. An edge e_{ij} can be either a *simple* or *conditional* dependency. An edge e_{ij} is a conditional dependency, i.e., $e_{ij} \in E_C$, if it will be taken based on a certain logical condition in the instruction I_i, i.e., for example, based on a "true" or "false" value of an *if* statement in I_i. If the edge e_{ij} is the only alternative for program execution, we will consider that it is a simple dependency, i.e., $e_{ij} \in E_S$. The embedded-application module M can be eventually implemented either is software or in hardware and will be referred as a *module* in the paper.

In Fig. 2a and b, we present a simplified example of a module M_1 and the corresponding dependency graph G_1. This module calculates a factorial of an integer number N, where N is an input, assigned to 5 for this particular example. Graph G_1 consists of 7 instructions, I_0 to I_6, and 7 dependency edges, e_{04} to e_{53}. Dependencies e_{36}, e_{34} and e_{53} constitute a while loop, i.e., "while i is less than or equals N", where edges e_{36} and e_{34} are conditional dependencies. Edge e_{36} is taken if the "while" condition I_3 is "false" and edge e_{34} is taken if the "while" condition I_3 is "true". Edge e_{53} is a simple dependency and is always taken after the last loop instruction I_5 to come back to the "while" instruction I_3. The dependency graph G_1 has three source nodes, I_0, I_1 and I_2, where I_0 is initialization of input variable N, and one sink node I_6.

We assign execution time to instructions in the module by means of application profiling. Execution time (ET) for each instruction (INSTR) of module M_1 is shown in Fig. 2c. For example, instruction I_4 takes 15 time units to execute. Similarly, time constraints, or *deadlines*, are assigned to the modules between two instructions. For example, module M_1 produces a new value of the *factorial* variable at each loop of I_3 to I_5 and each execution of I_3 to I_5 is constrained with the deadline D_1 of 50 time units, as depicted in Fig. 2b. Thus, instruction I_3 to I_5 are not allowed to execute more than 50 time units.

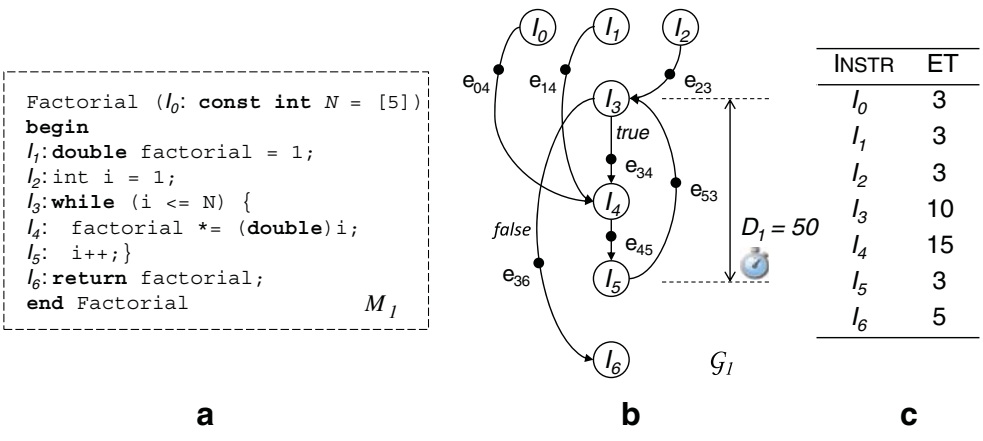

Fig. 2 Example of a module (**a**), its control-data-flow graph (**b**), and the execution-time associated with each instruction (**c**)

Since transient faults may affect execution of the application, we are interested in optimization of executable-assertion placement for the time-constrained module M_1.

3.2 B. Error Detection with Executable Assertions

Several soft errors can happen to the module M_1 in Fig. 2, due to transient faults:

- the multiplication operation may fail;
- the factorial number may be overflowed;
- the counter i may not increment as expected; and
- the while loop may loop infinitely due to a corrupted counter i, memory overflow, or problems with the conditional jump.

These errors can be triggered at any time of application execution and have to be detected. Several techniques can be used to detect errors caused by transient faults, such as watchdogs, signatures (both hardware and software), memory protection codes, various types of duplication, hardware-based error detections and, finally, assertions. In this work, we will use assertions to detect these faults in execution of module M_1.

Executable assertions are a common error detection technique, which is often used by programmers for debugging. In general, an assertion is a predicate written as a Boolean statement placed in the code, where the truth value should be always true in the absence of faults. An assertion can be defined as *if not* < assertion > *then* < error>, where < assertion > is a logical (Boolean) check of an operand value or correctness of an operation. An example of an operand assertion can be "a shall be 1". Correctness of an operation, for example, "$y=a-b$" can be checked with an assertion "$y-a+b$ shall be 0". An example of error manifestation, enclosed in the < error > block, can be a "try-catch" program structure or a dedicated error detection mechanism in form of macro definition that follows the assertion. The manifestation of error propagates to the external recovery or warning mechanism that initiates automatic recovery/restart or display a warning message to the user, i.e., a pilot that can initiate system reboot. Assertions can provide a very high level of error detection compared to other techniques since they can be fine-tuned to particular program properties.

However, assertions, similar to other error detection techniques, can introduce a significant performance overhead and, consequently, compromise the deadlines. At the same time, lack of assertions will lead to low error coverage and high susceptibility of the program to transient faults, which will not be detected and, hence, can lead to potentially catastrophic consequences. Thus, both performance overheads of assertions and their efficiency have to be considered in the assertion placement.

3.3 C. Parameters of Executable Assertions

To capture effectiveness of assertions, we assign to each assertion A_m a *tightness value* τ_m. The tightness value τ_m represents an increase in error detection probability of the module M against soft errors after assertion A_m is introduced. These values can be obtained with, for example, fault injection experiments [1] or with static probability analysis of the assertion code. We compute them as a part of our profiling strategy described in Section 4. We refer to this tightness value as the *static tightness*, which is the error detection capacity increase by the assertion in its respective instruction block taken in isolation. When the assertion is executed as the part of the program, its tightness value can be different. The tightness of the assertion will depend on the program execution pattern. We refer to the tightness value of assertions in the program as *dynamic tightness*, which is the error detection capacity increase by the assertion as a part of the program execution.

Each assertion A_m is also characterized with a *performance degradation* value, δ_m. The performance degradation value δ_m is the performance overhead of the assertion if introduced into module M. These values can be obtained with static analysis [57] or with extensive simulations of program execution [30]. We also compute them in the profiling step of the optimization framework, as described in Section 4. Respectively, we define *static-* and *dynamic-performance degradations*. The static-performance degradation is the overhead of an assertion associated with an instruction block taken in isolation; the dynamic-performance degradation is the overhead with respect to the program.

As a rule of thumb, assertions shall be introduced with the *highest tightness* at the *lowest performance degradation*.

Let us consider the example in Fig. 2. Instruction I_4: can be protected with assertion A_1 (where the assertion code is indicated with brackets):

```
<x = factorial;>
factorial *=(double)i;
<if (!(factorial/(double)i == x)) error();>
```

Assertion A_1.

For instruction I_4 this assertion protects only the multiplication operation, but it protects neither the value of factorial nor the value of the counter. Another assertion A_2 could be as follows:

```
<i_prev = i;>
<factorial_prev = factorial;>
<<<while loop iteration>>>
<x = factorial;>
factorial *=(double)i;
<if (!(factorial/(double)i == x
     && i_prev == i
     && factorial_prev == x) error();>
```

Assertion A_2.

This assertion A_2 will protect both the multiplication and the changing of counter *i* and *factorial* variables.

Let us consider that, after profiling, assertion A_1 gives tightness of 75 % (it captures faults only in the multiplication) and assertion A_2 gives tightness of 90 %.[1] Regarding performance, we obtain that A_1 has performance degradation of 20 time units, and A_2 has performance degradation of 30 time units. If we compare assertions A_1 and A_2 from the performance degradation point of view, we can see that assertion A_2 requires more time to execute. However, A_2 is better than A_1 from the tightness point of view.

Let us consider assertion A_3, which is the assertion A_2 excluding the assertion A_1 part for the multiplication check:

```
<i_prev = i;>
<factorial_prev = factorial;>
<<<while loop iteration>>>
<x = factorial;>
factorial *=(double)i;
<if (!(i_prev==i
     && factorial_prev==x)) error();>
```

Assertion A_3.

A_3 will give us 25 % tightness but will only need 10 time units to execute.

Note that assertions themselves can be subject to transient fault occurrences and, therefore, additional measures should be taken to address error detection in assertions. This could lead to a problem of "false positives", i.e., assertion affected by a transient fault can signal that a fault has happen but it actually has not. This can be solved with *self-detectable assertions*, i.e., we introduce assertions for assertions to provide a level of error detection coverage in the assertions' code. For example, a self-detectable assertion for assertion A_1 can be:

```
<if (!(factorial / (double)i == x))
   if (!((x * (double)i == factorial)))
     error();>
```

Example of self-detectable assertion.

[1] Note that these and the other values in the example are presented here for illustrative purposes only, i.e., in order to illustrate decision-making in the assertion placement process in the reader-friendly fashion.

This assertion checks if the division operation within assertion A_1 is performed correctly.

So, which assertion should we choose for module M_1 in Fig. 2, given a list of assertions A_1, A_2 and A_3 and the deadline D_1, i.e., 50 time units for instruction I_3 to I_5? The total execution without assertions will be for module M_1: 10+15+3=28, which gives a *performance budget* of 50−28=23 time units. We define the performance budget as the *difference between a given time constraint of the module and its actual time overhead of the module*. Assertion A_2 cannot be chosen since it executes in 30 time units, which exceeds the performance budget. Assertions A_1 and A_3 can both fulfill the performance requirement. However, assertion A_1 provides a better tightness value. Thus, assertion A_1 will be chosen since it has a performance degradation of 20 time units, which fits into the given budget, and its tightness value of 75 % is greater than 25 % tightness of A_3.

Although, for the example in Fig. 2, decision on which assertion to choose is relatively straightforward, as the size of application increases, these decisions become much more difficult. In the example of assertion selection in Fig. 2, we had to choose between three assertions for illustrative purposes. However, for real-life programs the number of assertions can be thousands, which makes it impossible to decide manually. On top of that, self-detectable assertions should be also considered to reduce the number of "false positives".

Another problem with assertions is that not all of the instructions are executed at every execution of an application. For example, in Fig. 3a an instruction I_2 will be executed only if the instruction I_1: "*if* x>99" produces "true" value. Suppose that the values of x are uniformly distributed between 1 and 100. Then, if we introduce an assertion A_m for instruction I_2, this assertion delivers its tightness only in 1 case out of 100. In 99 cases it does not contribute to the error detection of transient faults. We consider that, if, for example, the initial *static* tightness of A_m is 80 %, the actual *dynamic* tightness is 80 / 100=0.8 %. Thus, efficiency of assertions also depends on how often they (and their related instructions) are executed. In Fig. 3a, introduction of an assertion A_n with static tightness of 40 % into the "false" branch make more sense, i.e., its dynamic tightness is 40×99 / 100=39.6 %, which is greater than that of A_m. Note, however, that in this example, there is no competition between A_m and A_n as long as they are executed completely in different branches and both assertions can be introduced. Let us consider another situation depicted in Fig. 3b, where parts of A_m and A_n have to be executed before the condition I_1, i.e., always, with remaining parts to be completed in their own branches. If we have to choose between these assertions, assertion A_n is obviously the best, despite the fact that its static tightness (40 %) is only half of the static tightness (80 %) of A_m.

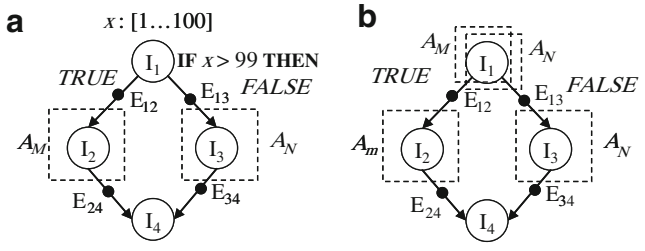

Fig. 3 Example of a conditional execution of an assertion

Thus, to address the complexity of the assertion placement, we have proposed the assertion-placement-optimization framework described in the following section.

4 Assertion-Placement Framework

We formulate the assertion-optimization problem as follows. As an input we get a module M of a time-constrained embedded application. Module M does not contain executable assertions. Several sets of instructions in this module M are associated with hard deadlines, as illustrated in the simple example of Fig. 2. A list of candidate assertions for this module M is also given. This list can be, for example, provided by designers after previous debugging of this module in the non-real-time mode or may even be associated with the module source code directly under the "_DEBUG" compilation flag. As an output, we want to produce a module with the subset of assertions introduced at the best possible places in the module source code, which maximize tightness, while meeting hard deadlines.

In this section, we present the approach for placement, optimization, and evaluation of error detection primitives. In particular, an overview of the developed framework is shown in Fig. 4. The framework is based on three main iterative phases: the code analysis and manipulation phase, the module simulation and profiling phase, and, finally, the optimization of the assertion placement according to the simulation and profiling information.

The framework takes as input a module (C/C++ code) of a time-constrained embedded system, which does not have any assertions, i.e. a fault silent description M, and transforms it into an intermediate representation [12]. In this phase, the framework introduces assertions, i.e. A, and placeholders by exploiting the dependency-graph associated with the module, i.e. G. Moreover, the framework allows the user to provide an actual assertion for each placeholder. Then, the module description with placeholders, i.e. M^L, is simulated for generating profiling information. In this work, we adopted a Monte-Carlo automatic-test-pattern generator (ATPG) for generating simulation stimuli, but either user-defined testbenches or structural-ATPG approaches

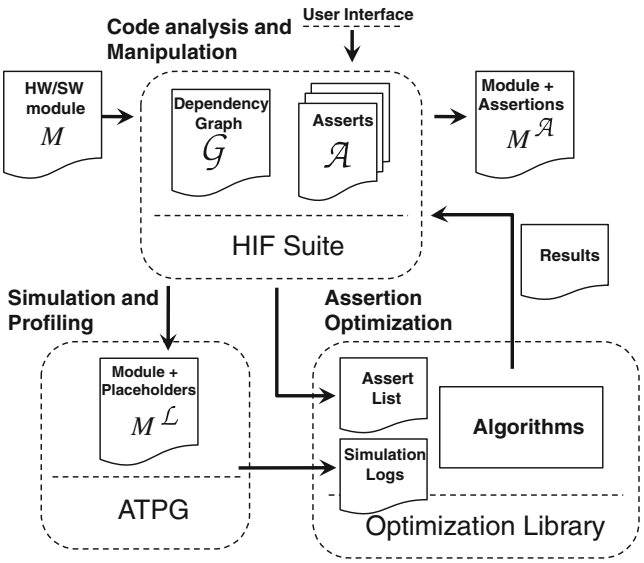

Fig. 4 The profiling and optimization framework for assertion placement

can be easily integrated. The generated profiling information is a simulation log which, for each assertion, contains the values of tightness and performance degradation. Finally, in the optimization phase, the framework exploits this information and the user-defined algorithms for generating a module description with assertions. In particular, the framework provides an API for accessing the static and dynamic profiling information. Moreover, it provides a set of optimization algorithms, which can be either used or extended by the designer. The final choice of assertions, i.e. M^A, is both compatible with the time constraints of the embedded application, and optimized in terms of error detection.

In the following, Section 4.A summarizes the main aspects of the code analysis; Section 4.B presents the assertion-profiling approach; Section 4.C describes the assertion-optimization infrastructure and some algorithms built on top of it; finally, Section 4.D provides an evaluation metric for validating the quality of the optimization environment.

4.1 A. Code Analisys and Profiling

The analysis and manipulation of the module descriptions are based on HIFSuite [12], a set of tools and libraries, which provides support for modeling and verification of embedded systems. The core of HIFSuite is an intermediate format, i.e., Heterogeneous Intermediate Format (HIF), which is similar to an abstract-syntax tree (AST); moreover, front-end and back-end tools allow the conversion of HW/SW description into HIF code and vice versa. In this initial phase, the HIF representation of the module is automatically converted into a dependency graph G, whose nodes and data/control-dependency edges are added to the HIF AST. Then, the code analysis searches for eligible locations for assertion placing. Eligible locations are assignment instructions, arithmetic expressions, control statement, operations over signals, bodies of loops, as well as initial and final instructions of processes. For each of these locations the framework provides a candidate assertion, which aims at detecting soft errors. For example, in the case of a conditional statement, it is necessary to guarantee that a transient fault does not affect the choice of the branch currently in execution, as shown in Listing 1.

Analogously, assertions may check the execution of the body of loop statements, or that the loop-counter monotonically increases (decreases). Another example is given in Listing 2, where both the array access and the data reading are protected against soft errors by means of an assertion. Moreover, a user interface permits the designer to introduce

Listing 1 Protecting conditional-statement branches against soft errors

```
1.  if (x <= max) {
2.    // then body
3.    <if (!(x <= max)) error();>
4.    x = y + z;
5.    // then body
6.  } else {
7.    // else body
8.    <if (x <= max) error();>
9.    // else body
10. }
```

Listing 2 Protecting array reading against soft errors

```
1.  <int _i;>
2.  x = a[_i = i];
3.  // ...
4.  <if ( x != a[i] && i != _i) error();>
```

further assertions or modify the assertions that the framework automatically choose.

During the analysis of the code, an initial statically-computed value for performance degradation is associated with each assertion. The performance degradation is computed based on the syntactic complexity of the Boolean predicate, which is defined by the number and type of variables and operators.

As a final step, in each eligible location, a placeholder is injected that registers an event in the simulation log every time it is reached during the execution. In particular, values for tightness and performance degradation are dynamically-computed and registered. An intuitive example of tightness computation for executable assertions is reported in Fig. 3. In such a case, the higher the number of times that a branch is executed, the greater is the tightness of each assertion that occurs in the branch. During the execution, the performance degradation, associated with each assertion, is computed as described in the following Section 4.B.

4.2 B. Executable-Assertion Profiling

Two critical performance measures characterize embedded applications: physical-resource requirements, e.g., memory occupation, and execution time of application tasks. In real-time applications, the latter is traditionally the most critical issue, indeed missing a deadline may cause result degradation or a complete failure.

In particular, our approach introduces additional code, i.e., executable assertions, which may increase the execution time and cause failures with respect to time constraints. In this context, a further clarification is necessary. In the design and verification practice, the use of executable assertions addressing design errors is limited to the pre-release phases of the life cycle of embedded application. When verification activities are completed and the application is ready to be released, the assertions are removed. Removing assertions after verification is analogous to compiling without the debug flags: the designer does not longer need to look for incorrect behaviors. On the contrary, in our approach executable assertions monitor constantly the application status for detecting soft errors after system deployment. Thus, measuring the introduced overhead is mandatory both for assertion-placement optimization and real-time requirements.

Execution-time profiling of assertions in embedded application, i.e., C code, encounters some obstacles:

```
1.  // tsc.h
2.  #include <stdio.h>
3.  __inline__ uint 64_t rdtsc(void) {
4.    uint32_t lo, hi;
5.    __asm__ __volatile__ ( // serialize
6.    "xorl %%eax, %%eax \n cpuid"
7.    ::: "%rax", "%rbx", "%rcx", "%rdx");
8.    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
9.    return (uint64_t)hi << 32 | lo;
10. }
```

Listing 3 Usage of RDTSC instruction at C-code level for x86-architecture. The instruction CPUID force in-order execution

```
1. rdtsc            ; read time stamp
2. move time, eax   ; move counter into variable
3. fdiv             ; floating-point divide
4. rdtsc            ; read time stamp
5. sub eax, time    ; find the difference
```

Listing 4 Usage of RDTSC instruction at assembly-code level. Out-of-the-order execution may alter the cycle count

(a) the optimization proposed in this work is mostly based on simulation, thus we prefer a dynamic-profiling approach; (b) modern processors have high execution frequencies and most of the available profiling tools does not have sufficient time accuracy, thus we need an approach providing accurate measures; at the same time, (c) the profiling activities typically introduces extra overhead, thus we need an approach with a reduced impact; finally, (d) we want to reduce the code instrumentation and, more in general, the designer intervention. The adopted solution addresses each of these issues, by exploiting two profiling techniques: time stamping and use of an explicit counter at a known frequency.

Nowadays, many processors provide dedicated synchronous counters, i.e., *time stamp counter* (TSC), for clock-cycle measurement [25]. TSCs share the same clock as the processor core and hence provide accuracy down to individual clock cycle of the operating frequency. For example, if the system is equipped with 1 GHz processor, the timing accuracy is 1 ns. Note that profiling tools like *GProf* [17] and *OProfile* [29], although widely used for general profiling of software, truncate results to milliseconds, thus for most of the assertions the resulting execution time is just *0 ms*. In contrary, a TSC-based approach is useful when a very fine cycle measurement is needed for a small section of code: it can give a good idea of how many cycles a few instructions might take versus another set of instructions.

The TSC register can be accessed using the assembly instruction *read time stamp counter* (RDTSC). Listing 3 provides the snippet of C/assembly code that can be used for profiling on *RTLinux* and *x86* architecture, but similar code can be written for different architectures. For example, the *ARM11* and *Cortex* cores include a Performance Monitor Unit for events counting. Note that, in Listing 3 at lines 5–7, *instruction serialization* is introduced. Indeed, recent processors support *out-of-order execution* that is instructions are not necessary performed in the order they appear in the source code. This may give a misleading cycle count. An example is reported in Listing 4: the assembly code fragment measures the time required by a floating point division (line 3). Since the floating-point division takes a longer time to compute, the following instruction RDTSC could actually execute before the division. For example, under the hypothesis of out-of-the-order-execution, a possible re-ordering of the instructions in Listing 4 is <1, 2, 4, 3, 5>. If this happened, the cycle count would not take the division into account. In order to keep the RDTSC instruction from being performed out-of-the-order, the instruction CPUID is used. CPUID forces every preceding instruction in the code to complete before allowing the application to continue. Note that CPUID is normally used to identify the processor on which the application is being run, but here it is used only to force the in-order execution.

Listing 5 provides a simple example of usage of the function defined in Listing 3. Since the TSC counts cycles, in order to convert the clock ticks to time we need to use the classical equation: #seconds = #cycles / (frequence Hz). On the same example, GProf is not able to provide a result for the assertion at line 8.

Finally, dynamic profiling of executable assertions in embedded applications may produce different results due to short-circuit operators and cache effects.[2] The short-circuit evaluation of Boolean operators in C specifies that the left operand is evaluated first; then, depending on the result of the evaluation of the left operand, the right operand is evaluated. For example, let us consider the expression `((a!=0)&&((b+4)>0))`. If the variable `a` stores the value `0`, then the overall Boolean expression is trivially false and the sub-expression `((b+4)>0)` is not evaluated. This may mislead the execution-time measurement of the assertion. Similarly, when an assertion is evaluated and the first time a line of data is brought into cache, it takes a large number of cycles to transfer data from the main memory.

[2] In real-time embedded systems, due to difficulties of obtaining reliable predictions, caching mechanisms are frequently disabled.

```
1.  #include <assert.h>
2.  #include <stdio.h>
3.  #include "tsc.h"
4.  int main (int argc, char** argv) {
5.  srand ((unsigned)time(NULL));
6.  int r = rand() % 100;
7.  uint64_t before = rdtsc();
8.  assert(r!=0);
9.  uint64_t after = rdtsc();
10. uint64_t diff = after - before;
11. printf("%llu\n", diff);
12. }
```

Listing 5 Time stamping and use of TSC counter permit to measure accurately assertion overhead

For addressing these issues we adopt a *cache-warming* technique and an *on-the-site-measure* hypothesis. In particular, the embedded application is executed for a long time with various input stimuli and for each assertion average execution time is computed. This does not remove all the effects of cache and short-circuits operators, but provide a close measure of performance degradation due to executable assertions. In particular, cache missing will occur differently on different iteration by producing a different cycle count; similarly, the Boolean expression will be differently exercised. This approximates the behavior of the embedded application and assertions after system deployment (i.e., on-the-site-measure hypothesis).

4.3 C. Optimization Infrastructure

In the previous phases, a set of candidate assertions addressing soft errors is associated with each module of the time-constrained embedded application, and simulation-based profiling information is generated. These assertions have different probabilities of detecting errors, and increase by different amounts the execution time of the module. The proposed framework provides an infrastructure that allows the designer to automatically choose the assertions that maximize the error detection and respect the time constraints of the application.

Algorithm 1 The Slowest Assertion First (SAF) algorithm

```
1.  input: the module M
2.  input: the set CAS of candidate assertions
3.  input: the maximum tolerated overhead MTO
4.  output: the set SAS of selected assertions

5.  SAS ← ∅; remaining_overhead ← MTO;
6.  APQ ← performance_degradation_ordering (CAS);
7.  while ((remaining_overhead ≥ 0) ∧ APQ.is_not_empty()) do
8.  assertion ← remove_top (APQ);
9.  if (assertion.perf_degradation ≤ remaining_overhead) then
10. SAS ← SELECTED ∪ { assertion };
11. remaining_overhead ←
            remaining_overhead - assertion.perf_degradation;
12. end if
13. end while
14. return SAS
```

Table 1 ITC'99 benchmarks characteristics

Bench.	Pi	Po	Var	Loc	CAS	OH (ns)
b01	4	2	3	186	54	217
b02	3	1	3	131	28	117
b03	6	1	26	212	68	308
b04	13	8	101	194	55	234
b05	3	36	511	362	192	1622
b06	4	6	3	205	67	288
b07	3	8	43	286	46	195
b08	11	4	37	137	29	128

The optimization library provides functionalities for accessing the dependency and profiling information. In particular, some data structures are maintained in association with the dependency graph of each module, for example:

- A *set* which contains each *candidate assertion* (A_m), the related location in the module (l_m), and the profiled tightness (τ_m), and performance degradation (δ_m).
- A *set of paths* which contains each path followed during the simulation. In particular, a path is represented as a list of events in the simulation log, and an event is a couple (A_m, t), where A_m is a reached assertion and t is the corresponding execution time from the simulation beginning.

We implemented some algorithms on top of this infrastructure, with the purpose of showing possible assertion-placing optimizations, and focusing the attention on the usability of the library. In particular, we can distinguish between algorithms which use only the structural information of the module, and algorithms that exploit the simulation information.

A first simple optimization approach is the *Slowest Assertion First* (SAF) strategy, shown in Algorithm 1. It always chooses, while the deadlines are respected, the candidate assertion with the *highest static-performance degradation*. In such a case, we assume that the performance degradation is proportional to the error detection of the assertions; that is, a computational-heavy assertion detects more likely soft errors. The expected result of this algorithm is a design with few, but effective, assertions. In particular the algorithm takes as inputs (lines 1–3) the module *M*, the set of candidate assertions *CAS*, and a maximum-tolerated-overhead value *MTO*, which depends on the time constraints of the embedded system. It returns a set of assertions *SAS* (line 4), which are compatible with the constraints and aim at optimizing the soft-error detection probability. In particular, the candidate assertions are ordered with respect to the static performance degradation, with the most expensive first, in a priority queue *APQ* (line 6). Then, the assertions are selected till they exhaust the available tolerated overhead (lines 7–13).

Another similar approach, but based on a conceptually inverse motivation, is the Fastest Assertion First (FAF). It always chooses the available assertion with the minimum performance degradation, until the tolerated overhead is exhausted. The expected result is a design with the highest number of assertions, which try to maximize the fault detection capability. A third algorithm, the Most Executed Assertion First (MEAF), exploits the tightness of the assertions, a simulation-based information: it chooses, while the deadlines are respected, the most executed assertion during the system simulation. Similarly, several other algorithms can be easily created by exploiting and combining both the structural and profiling information.

Table 2 UniVR benchmark characteristics

Bench.	Pi	Po	Var	Loc	CAS	OH (ns)
root	33	33	141	90	44	188
ecc1	23	32	85	312	52	192
8b10b	9	10	37	455	82	358
adpcm	128	67	286	277	100	439
dist	66	33	4096	190	104	429
div	33	33	144	261	133	576
tcas	22	17	121	1415	222	940
aftt	132	65	1713	1813	636	3077

Table 3 Comparison of the optimization algorithms (b05 benchmark)

		SAF		FAF		Meaf	
MTO (%)	MTO (ns)	Ratio	SAS	Ratio	SAS	Ratio	SAS
5	81.1	0.182	8	0.027	16	0.289	10
10	162.2	0.192	14	0.075	34	0.426	18
15	243.3	0.295	21	0.125	51	0.557	28
20	324.4	0.407	50	0.317	67	0.637	41
25	405.5	0.448	36	0.359	80	0.704	56

Table 4 Comparison of the optimization algorithms (aftt benchmark)

		SAF		FAF		Meaf	
MTO (%)	MTO (ns)	Ratio	SAS	Ratio	SAS	Ratio	SAS
5	153.9	0.256	30	0.001	40	0.390	38
10	307.7	0.122	64	0.390	71	0.756	68
15	461.6	0.500	93	0.512	103	0.851	98
20	615.4	0.122	121	0.634	139	0.874	132
25	769.3	0.366	150	0.756	169	0.910	161

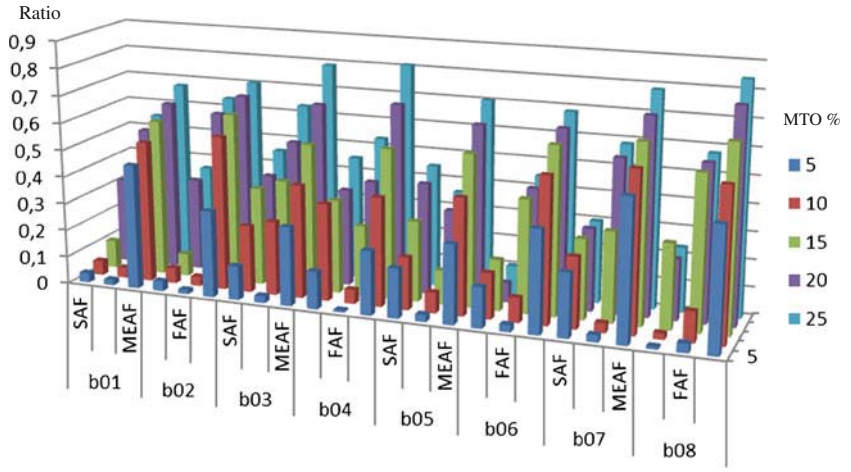

Fig. 5 Comparison of the optimization algorithms for ITC'99 benchmarks

4.4 D. Evaluation Metric

Measuring the quality of selected assertions is a key aspect for evaluating assertion-optimization algorithm. Since the framework is particularly focused on the simulation, it seems reasonable to emphasize simulation contribution also for evaluating the obtained results. Thus, we propose the following metric.

Definition 1 Let M be a time-constrained module, $A = \{A_1, \ldots, A_K\}$ a list of candidate assertions for the module M, and p an assertion-placing algorithm. Then the result of the algorithm p over the module M, and the set of assertions A is the set of assertions $A_p = \{A_1^p, \ldots, A_H^p\} \subseteq A$. The quality of A_p is measured as the ratio

$$Ratio_p = \frac{\sum_{i=1}^{H} \frac{\tau_i^p}{\delta_j^p}}{\sum_{j=1}^{K} \frac{\tau_j}{\delta_j}}$$

where $0 \leq \tau_i \leq 1$ and $\delta_i \geq 0$ are, respectively, the tightness and performance degradation associated with each assertion A_j in A, while $0 \leq \tau_i^p \leq 1$ and $\delta_i^p \geq 0$ are associated with each assertion A_i^p in A_p.

This metric promotes the set of assertions having higher tightness and lower performance degradation. We empirically measured the tightness by using a transient fault simulator, i.e., FEMU [18]. The simulator permits to randomly inject bit faults in all general purpose registers and into the status flags of the emulated processor. Furthermore, it is possible to manipulate the first two bytes of a fetched instruction, in order to simulate the execution of a wrong instruction. In particular, the duration of the fault can be configured to be permanent or transient. The performance degradation is measure in terms of time overhead introduced by the assertions, as described in Section 4.B.

5 Experimental Results

In order to assess the effectiveness of the proposed framework, we have used the benchmarks described in Tables 1 and 2. In each table, columns PI, PO, and VAR denote

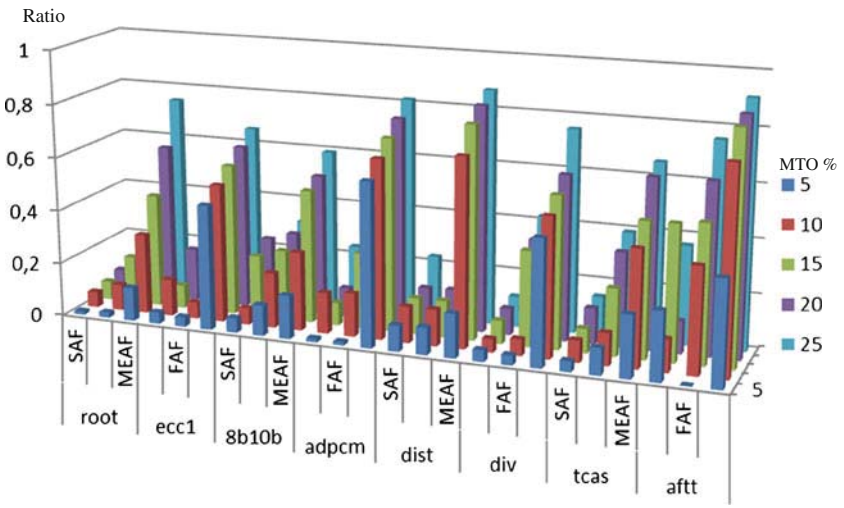

Fig. 6 Comparison of the optimization algorithms for UniVR benchmarks

respectively the number of bits in primary inputs, primary outputs and internal variables for each benchmark; column LOC reports the number of lines of code; column CAS reports the number of candidate assertions; finally, column OH reports the overall overhead in nanoseconds (ns) that the candidate assertions introduce. Benchmarks in Table 1 are from ITC'99 suite, which is a well know reference used by other authors [14]. In particular, we have rewritten the behavioral designs in C code to be executed as embedded applications. Indeed, the designs are suitable to be implemented as software components: for example, design *b03* is a resource arbiter and *b10* is a voting system. Table 2 reports a further set of embedded applications, i.e., UniVR suite, which we used in our experiments. In this case, the applications are ordered according the number of candidate assertions (column CAS). In particular, *root*, *div*, and *dist* are software components of an industrial face recognition system provided by STMicroelectronics; *ecc1* is an error-correction application by STMicroelectronics; *8b10b* is a encoding/decoding application [39]; *adpcm* is implementing an audio-compression algorithm by STMicroelectronics; *tcas* is a traffic-collision-avoidance system [53]; finally, *aftt* computes the trajectory of aircraft-fuel tanks released before emergency landing.

After the automatic generation of a set of candidate assertions for each benchmark, we profiled the quality of the described optimization algorithms by using the evaluation metric proposed in Section 4.D. In particular, we adopted a Monte-Carlo automatic-test-pattern generator for generating simulation stimuli. The embedded applications are executed for a long time with various input stimuli and for each assertion average values for tightness and performance degradation are computed. We observed that different runs, i.e., different workloads, provide approximately the same results (the difference of tightness and performance degradation are under 1 %).

In Tables 3 and 4, we report the results of the algorithms in columns SAF, FAF, and MEAF for benchmarks *b05* and *aftt,* respectively. We have chosen one benchmark for each set of embedded application having highest number of candidate assertions, i.e., *b05* and *aftt*. We executed each algorithm with a maximum-tolerated overhead. Such a value is expressed as a percentage of the overall overhead reported in column OH(ns) of Tables 1 and 2. In particular, columns MTO(%) and MTO(ns) of Tables 3 and 4 report the percentage value and the corresponding time value in nanoseconds, respectively. Moreover, for each algorithm, the column RATIO reports the quality of the algorithm which is measured according to the proposed metric, and column SAS reports the number of assertion selected over the total number of candidate assertions reported in Tables 1 and 2.

According to the results we can observe that:

- there is an increasing in the quality ratio of the selected assertions with the increase of the maximum tolerated deadline; indeed this is expected, because, apart from the adopted optimization algorithm, an higher number of assertions can be selected for each benchmark;
- the number of selected assertions of the algorithm Fastest Assertion First (FAF) is always higher than Slowest Assertion First (SAF), but this does not always guarantee an higher effectiveness of the assertions;
- the Most Executed Assertion First (MEAF) algorithm always outperforms the others; this result highlights the importance of the simulation and profiling information in selecting the candidate assertions; indeed, the proposed framework provides an infrastructure to define effective optimization algorithms, which both increase the efficiency of assertion checks and reduce the performance overhead.

Figures 5 and 6 provide an overview of the quality (Ratio) for each benchmark and algorithm. The maximum-tolerated overhead (MTO) is expressed as a percentage ranging from 5 % to 25 %. From these results we can further conclude that for each benchmark the profiling/simulation-based approach (MEAF) provides better results than the static-analysis-based optimizations. The mean effectiveness of MEAF is 3 times higher the others, while, in the case of benchmark *b01*, the effectiveness of MEAF is up to 9 times higher. Moreover, the variance[3] of the MEAF results ($0.2448*10^{-2}$) is significantly smaller than SAF ($1.1714*10^{-2}$) and FAF ($2.9704*10^{-2}$) that guarantees a higher confidence on the quality of the error-detection primitives which are selected by means of the profiling framework.

6 Conclusion

Despite the intensive efforts of engineers during the design and verification phases, time-constrained embedded applications may still have erroneous behaviors after deployment. Some of these "buggy" application behaviors could be triggered by the unavoidable nature of transient faults, caused by cosmic radiations, temperature variations, vibrations, interferences, etc. Some could be the result of aging and some could be purely software and/or hardware related due to, for example, a rare synchronization circumstance or sudden incorrect initialization of a certain variable. In this context, executable assertions can be inserted into the application code to constantly monitor the applications behavior, as long as they respect the time constraints of the applications. In this work, we have presented an approach to introduce executable

[3] The variance describes how far values lie from the mean.

assertions into time-constrained embedded applications. The approach takes into account frequency of assertion execution, tightness, i.e., error-coverage efficiency, of the assertions and performance degradation due to them.

We have also developed an optimization framework for assertion placement in embedded applications given a set of candidate assertions. In particular, it identifies candidate locations for assertions and associates a candidate assertion to each location. The selected assertions will respect the time-constraints of the real-time embedded application. The proposed framework considers both assertion properties and properties of real-time applications by accurately profiling them. Our experimental results have shown that we could introduce executable assertions and increase error detection probabilities against transient faults, while, at the same time, preserving timing constraints. Thus, executable assertions can be effectively used to improve error-detection and debuggability of real-time applications.

References

1. Aidemark J, Vinter J, Folkesson P, Karlsson J (2001) GOOFI: generic object-oriented fault injection tool. Proceedings of International Conference on Dependable Systems and Networks, pp 83–88
2. Alkhalifa Z, Nair VSS, Krishnamurthy N, Abraham JA (1999) Design and evaluation of system-level checks for on-line control flow error detection. IEEE Trans Parallel Distr Syst 10(6):627–641
3. Ando H, Yoshida Y, Inoue A, Sugiyama I, Asakawa T, Morita K, Muta T, Motokurumada T, Okada S, Yamashita H, Satsukawa Y, Konmoto A, Yamashita R, Sugiyama H (2003) A 1.3 GHz fifth generation SPARC64 Microprocessor. Proceedings of International Solid-State Circuits Conference
4. Austin TM (1999) DIVA: a reliable substrate for deep submicron microarchitecture design. Proceedings ACM/IEEE International Symposium on Microarchitecture. IEEE Computer Society, pp 196–207
5. Ayav T, Fradet P, Girault A (2008) Implementing fault-tolerance in real-time programs by automatic program transformations. ACM Trans Embed Comput Syst 7(4):1–43
6. Baleani M, Ferrari A, Mangeruca L, Sangiovanni Vincentelli A, Peri M, Pezzini S (2003) Fault-tolerant platforms for automotive safety-critical applications. Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp 170–177
7. Baum CE (1992) From the electromagnetic pulse to high-power electromagnetics. Proc IEEE 80(6):789–817
8. Baumann RC (2001) Soft errors in advanced semiconductor devices—part I: the three radiation sources. Device and Materials Reliability, IEEE Transactions, vol. 1, no. 1
9. Baumann RC (2002) Soft errors in commercial semiconductor technology: overview and scaling trends. Proceedings of Reliability Physics Tutorial Notes, Reliability Fundamentals
10. Baumann RC (2005) Radiation-induced soft errors in advanced semiconductor technologies. IEEE Transactions on Device and Materials Reliability, vol. 5, no. 3
11. Benso A, Chiusano S, Prinetto P, Tagliaferri L (2000) A C/C++ source-to-source compiler for dependable applications. Proceedings of IEEE International Conference on Dependable Systems and Networks, pp 71–78
12. Bombieri N, Di Guglielmo G, Fummi F, Pravadelli G, Ferrari M, Stefanni F and Venturelli A (2010) HIFSuite: tools for HDL code conversion and manipulation. EURASIP Journal on Embedded Systems, vol. 2010
13. Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Sonza Reorda M, Violante M (2000) Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. IEEE Trans Nucl Sci 47(6):2231–2236
14. Corno F, Reorda M, Squillero G (2000) RT-Level ITC99 Benchmarks and First ATPG Result. IEEE Design & Test of Computers, pp 44–53, July–August
15. Eles P, Peng Z, Pop P, Doboli A (2000) Scheduling with bus access optimization for distributed embedded systems. IEEE Trans VLSI Syst 8(5):472–491
16. Engel H (1997) Data flow transformations to detect results which are corrupted by hardware faults. Proceedings of IEEE High-Assurance System Engineering Workshop, pp 279–285
17. Fenlason J, Stallman R (1998) GNU GProf. GNU Free Software Foundation
18. Gaiswinkler G, Gerstinger A (2009) Automated software diversity for hardware faul detection. Proceedings of IEEE Conference on Emerging Technologies and Factory Automation
19. Gill B, Seifert N, Zia V (2009) Comparison of alpha-particle and neutron-induced combinational and sequential logic rates at the 32 nm technology node. Proceedings of IEEE International Reliability Physics Symposium, pp 199–205
20. Goloubeva O, Rebaudengo M, Sonza Reorda M, Violante M (2003) Soft-error detection using control flow assertions. Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp 581–588
21. Goloubeva O, Rebaudengo M, Sonza Reorda M, Violante M (2006) Software-implemented hardware fault tolerance. Springer
22. Hiller M (2000) Executable assertions for detecting data errors in embedded control systems. Proceedings of International Conference on Dependable Systems and Networks, pp 24–33
23. Hiller M, Jhumka A, Suri N (2002) On the placement of software mechanisms for detection of data errors. Proceedings of International Conference on Dependable Systems and Networks, pp 135–144
24. Huang KH, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. IEEE Trans Comput 33:518–528
25. Intel Corporation (1997) Using the RDTSC instruction for performance monitoring. Technical report
26. Koren I, Mani Krishna C (2007) Fault-tolerant systems. Elsevier
27. Lantz L (1996) Soft errors induced by Alfa particles. IEEE Trans Reliab 45:175–179
28. Levenson NG, Turner CS (1993) An investigation of the Therac-25 accidents. IEEE Comput 26(7):18–41
29. Levon J, Elie P (2005) OProfile: a system profiler for Linux. Web site: oprofile.sourceforge.net
30. Lu Y, Nolte T, Kraft J, Norstrom C (2010) Statistical-based response-time analysis of systems with execution dependencies between tasks. Proceedings of IEEE International Conference on Engineering of Complex Computer Systems, pp 169–179
31. Messenger G, Ash M (1986) The effects of radiation on electronic systems. Van Nostrand Reinhold Company Inc
32. Nicolaidis M (ed) (2010) Soft errors in modern electronic systems. Springer
33. Normand E (1996) Single event upset at ground level. IEEE Trans Nucl Sci 43(6):2742–2750
34. Oh N, McCluskey EJ (2002) Error detection by selective procedure call duplication for low energy consumption. IEEE Trans Reliab 51(4):392–402

35. Oh N, Mitra S, McCluskey EJ (2002) ED4I: error detection by diverse data and duplication instructions. IEEE Trans Comput 51:180–199
36. Oh N, Shirvani PP, McCluskey EJ (2002) Error detection by duplicated instructions in superscalar processors. IEEE Trans Reliab 51(1):63–75
37. Oh N, Shirvani PP, McCluskey EJ (2002) Control-flow checking by software signatures. IEEE Trans Reliab 51(2):111–122
38. Omana M, Rossi D, Metra C (2004) Latch susceptibility to transient faults and new hardening approach. IEEE Trans Comput 56:1255–1268
39. OpenCores (8b10b) Encoder/Decoder. www.opencores.org
40. Peti P, Obermaisser R, Kopetz H (2005) Out-of-norm assertions. Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium, pp 209–223
41. Piotrowski A, Makowski D, Jablonski G, Napieralski A (2008) The automatic implementation of software implemented hardware fault tolerance algorithms as a radiation-induced soft errors mitigation technique. Proceedings of IEEE Nuclear Science Symposium Conference Record, pp 841–846
42. Pradhan DK (ed) (1986) Fault-tolerant computing: theory and techniques. Prentice-Hall
43. Pradhan DK (1996) Fault-tolerant computer system design. Prentice Hall PTR
44. Rashid F, Saluja KK, Ramanathan P (2000) Fault tolerance through re-execution in multiscalar architectures. Proceedings of International Conference on Dependable Systems and Networks, pp 482–491
45. Rebaudengo M, Sonza Reorda M, Torchiano M, Violante M (1999) Soft-error detection through software fault-tolerance techniques. Proceedings of International Symposium on Defect and Fault Tolerance in VLSI Systems, pp 210–218
46. Rebaudengo M, Sonza Reorda M, Torchiano M, Violante M (1999) Soft-error detection through software fault-tolerance techniques. Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp 210–218
47. Rebaudengo M, Sonza Reorda M, Torchiano M, Violante M (2001) A source-to-source compiler for generating dependable software. Proceedings of IEEE International Workshop on Source Code Analysis and Manipulation, pp 33–42
48. Reinhardt SK, Mukherjee SS (2000) Transient fault detection via simultaneous multithreading. Proceedings of International Symposium on Computer Architecture, pp 25–36
49. Reis GA, Chang J, Vachharajani N, Rangan R, August DI (2005) SWIFT: software implemented fault tolerance. Proceedings of International Symposium on Code Generation and Optimization, pp 243–254
50. Rossi D, Omana M, Metra C (2010) Transient fault and soft error on-die monitoring scheme. Proceedings of International Symposium on Defect and Fault Tolerance in VLSI Systems, pp 391–398
51. Rotenberg E (1999) AR-SMT: a michroarchitecture approach to fault tolerance in microprocessors. Proceedings of International Symposium on Fault-Tolerant Computing, pp 84–91
52. Sangiovanni Vincentelli A, Di Natale M (2007) Embedded system design for automotive applications. Proc Comput 40 (10):42–51
53. Software-artifact Infrastructure Repository. Traffic-Collision-Advoidance system (TCAS). sir.unl.edu
54. Sohi G, Franklin M, Saluja K (1989) A study of time-redundant fault tolerance techniques for high-performance pipelined computers. Proceedings of International Symposium on Fault Tolerant Computing, pp 463–443
55. Vemu R, Abraham JA (2006) CEDA: control-flow error detection through assertions. Proceedings of IEEE International On-Line Testing Symposium
56. Voas JM, Miller KW (1994) Putting assertions in their place. Proceedings of International Symposium on Software Reliability Engineering, pp 152–157
57. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand R, Heckmann C, Mueller F, Puuat I, Puschner P, Staschulat J, Stenström P (2008) The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans Embed Comput Syst 7(3)
58. Wood KS, Fritz G, Hertz P, Johnson WN, Lovelette MN, Wolff MT, Bloom E, Godfrey G, Hanson J, Michelson P, Taylor R, Wen H (1994) The USA experiment on the ARGOS satellite: a low cost instrument for timing x-ray binaries. Proc EUV, X-Ray, and Gamma-Ray Instrum Astron 2280:19–30
59. Yau S, Chen F (1980) An approach to concurrent control flow checking. IEEE Trans Softw Eng SE-6(2):126–137
60. Yeh Y (1996) Triple-triple redundant 777 primary flight computer. Proc IEEE Aero Appl Conf 1:293–307
61. Yin H, Bieman JM (1994) Improving software testability with assertion insertion. Proceedings of International Test Conference, pp 831–839
62. Zenha Rela M, Madeira H, Silva JG (1996) Experimental evaluation of the fail-silent behavior in programs with consistency checks. Proceedings of Symposium on Fault Tolerant Computing, pp 394–403
63. Ziegler JF et al (1996) IBM experiments in soft fails in computer electronics (1978–1994). IBM J Res Dev 40(1):3–18

Viacheslav Izosimov is a Systems Architect Consultant at the Semcon AB corporation. He performs an advanced consultancy work in the area of safety-critical embedded systems, functional safety and reliability. In particular, he works with ISO 26262 and IEC 61508 standards. Viacheslav defended his PhD in Computer Systems at Linkoping University in 2009. He is author of more than 30 conference publications, several journal manuscripts and book chapters. He received the Best Paper Award at the Design, Automation and Test in Europe Conference (DATE 2005) and the EDAA Outstanding Dissertation Award 2011 for his PhD thesis work.

Giuseppe Di Guglielmo received the Laurea degree in Computer Science in 2005 and the PhD degree in Computer Science in 2009, both from the University of Verona. He has been a research assistant at the Computer Science Department of the University of Verona since 2009. His research interests include verification of embedded systems and EDA methodologies for hardware/software system modeling. He is a member of the IEEE.

Michele Lora received the Laurea degree in Computer Science from the University of Verona in 2010. Thesis title: "Profiling and simulation-based insertion and optimization of fault tolerance assertions in embedded time-constrained systems" developed under the supervision of Prof. Graziano Pravadelli and Dr. Giuseppe Di Guglielmo of University of Verona, in collaboration with the ESLab of the Linkoping Institute of Technology.

Graziano Pravadelli received the Laurea degree and the PhD degree in Computer Science from the University of Verona, respectively, in 2001 and 2004. He was assistant professor at the Computer Science Department of the University of Verona from January 2005 to December 2010, where, since January 2011 he is associate professor. His main research interests are in hardware description languages and electronic design automation methodologies for modeling and verification of hardware/software systems, in which area he has published about 80 papers. He is a member of the IEEE.

Franco Fummi received the Laurea degree in Electronic Engineering in 1990 and the PhD degree in Electronic Engineering in 1995, both from the Politecnico di Milano. He has been a full professor in the Dipartimento di Informatica of the University of Verona since 2001. His main research interests are in hardware description languages and electronic design automation methodologies for modeling, verification, testing, and optimization of hardware/software systems. He has published more than 230 papers in the EDA field; three of them received "best paper awards" at, respectively, IEEE EURODAC '96, IEEE DATE '99, FDL '11. Since 1995, he has been with the Dipartimento di Elettronica e Informazione of the Politecnico di Milano with the position of assistant professor. In July 1998, he was promoted to the position of associate professor in computer architecture in the Department of Computer Science at the University of Verona. He is a member of the IEEE and a member of the IEEE Test Technology Committee.

Zebo Peng has been Professor of the chair of Computer Systems and Director of the Embedded Systems Laboratory (ESLAB) at Linkoping University since 1996. He is also Chairman of the Division for Software and Systems (SaS). He served as Director of the Swedish National Graduate School in Computer Science (CUGS) in 2006–2008. He received his Ph.D. in Computer Science from Linkoping University in 1987. Prof. Peng's current research interests include design and test of embedded systems, electronic design automation, SoC testing, fault tolerant design, hardware/software co-design, and real-time systems. He has published over 300 technical papers in these areas and coauthored four books. Prof. Peng received "four best paper awards", two at the European Design Automation Conferences (EURO-DAC'92 and EURO-DAC'94), one at the IEEE Asian Test Symposium (ATS'02), and one at the Design, Automation and Test in Europe Conference (DATE'05). Prof. Peng served as the Chair of the IEEE European Test Technology Technical Council (ETTTC) in 2006–2009, and has been a Golden Core Member of the IEEE Computer Society since 2005.

Masahiro Fujita received his Ph.D. degree in Information Engineering from the University of Tokyo in 1985. From 1985 to 2000, he worked at Fujitsu Laboratories Japan and US. In March 2000, he joined the department of Electronic Engineering in the University of Tokyo as a professor, and now a professor at VLSI Design & Education Center in the University of Tokyo. He has received several awards from Japanese major scientific societies on his works in formal verification and logic synthesis.