Optimization of Assertion Placement in Time-Constrained Embedded Systems

Viacheslav Izosimov
Embedded Intelligent Solutions (EIS) By Semcon AB
Email: viacheslav.izosimov@eis.semcon.com

Michele Lora, Graziano Pravadelli, Franco Fummi
Dept. of Computer Science, University of Verona
Email: {name.surname}@univr.it

Zebo Peng
Dept. of Computer and Information Science,
Linköping University
Email: zebo.peng@liu.se

Giuseppe Di Guglielmo, Masahiro Fujita
VLSI Design and Education Center, University of Tokyo,
CREST - Japan Science and Technology Agency
{gdg, fujita}@cad.t.u-tokyo.ac.jp

Abstract— **We present an approach for optimization of assertion placement in time-constrained HW/SW modules for detection of errors due to transient and intermittent faults. During the design phases, these assertions have to be inserted into the executable code and, hence, will always be executed with the corresponding code branches. As the result, they can significantly increase execution time of a module, in particular, contributing to a much longer execution of the worst case, and cause deadline misses. Assertions have different characteristics such as tightness (or "local error coverage") and execution latency. Taking into account these properties can increase efficiency of assertion checks in time-constrained embedded HW/SW modules. We have developed a design optimization framework, which (1) identifies candidate locations for assertions, (2) associates a candidate assertion to each location, and (3) selects a set of assertions in terms of performance degradation and assertion tightness. Experimental results have shown the efficiency of the proposed techniques.**

I. Introduction

Executable assertions are one of the most efficient methods to increase testability of applications against transient and intermittent faults[1] (also known as "soft errors"). Transient faults are one of the most common faults in modern electronic systems due to high complexity, smaller transistor sizes, higher operational frequency, and lower voltage levels [4][12][19]. These faults can be result of electromagnetic interference, radiation, temperature variations, etc. [16]. They happen for a short time and disappear without causing a permanent damage to the circuit. However, if not tolerated, transient faults may crash the system or lead to dramatic quality deterioration [16][17]. Efficient error detection is an ultimate prerequisite for this fault tolerance. Assertions can provide a high degree of error coverage against transient faults with low performance overhead compared to other techniques [10][13][20]. This is, however, true only if the right assertions are introduced in the proper places during execution of HW/SW modules.

In this paper, we will focus on optimization of executable assertions placement in HW/SW modules. In particular, we will consider HW/SW modules, which have timing constraints, i.e., are parts of a larger real-time embedded system. Execution of such a real-time module has to complete before a certain deadline [18]. If this deadline is violated, it may lead to potentially catastrophic consequences for an application. Thus, taking into account performance overheads of executable assertions becomes particularly crucial.

The related works on assertions can be classified in three main groups: assertion-based approaches for SystemC code (HW/SW modules), assertion-based debugging approach for embedded applications, and assertion-based approaches for addressing transient faults.

In the first group of techniques, executable assertions are introduced into SystemC descriptions, which is a popular language for developing embedded systems [9]. In particular, Habibi et al. [11] have presented a methodology to generate assertions with ASM (Abstract States Machines) in C# language with the following transition into SystemC before integration into the HW/SW module. They use PLS (Property Specification Language) to specify a variety of design properties in ASM. Economakos [6] has proposed a framework to introduce SystemC and PSL assertions into embedded programs, which are later integrated into the high-level synthesis (HLS) process for digital circuits. Tomasena et al. [21] have proposed a design framework to introduce assertions into SystemC programs at the Transaction Level (TL) model with the Assertion Based Verification (ABV) support. ABV is a popular technique to verify electronic systems, where assertions are introduced in a systematic manner into an embedded component.

In the second group, most of assertion-related works target debuggability and testability of applications. For example, Yin and Bieman [15] have developed a C-Patrol tool to automatically introduce executable assertions into C programs. Voas and Miller [23] have introduced 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. Gharehbaghi et al. [8] have proposed an assertion-based methodology to test Systems-on-Chip (SoC) with a monitor that observes status of assertions.

In the last group of related research works, assertions have been used against transient faults. Vemu and Abraham [22] have proposed a CEDA methodology to call program flow with assertions, which can be used to detect transient faults. Goloubeva et al. [10] have proposed an approach to insert executable assertions for detecting transient faults. Hiller [13] has proposed 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 [14]. Peti et al. [20] have proposed an "out-of-norm" assertion methodology to insert assertions into electronic components, in particular, communication controllers, to detect transient faults. Ayev et al. [2] have proposed a technique to integrate assertions into embedded programs with automatic program transformations to provide error detection and recovery against transient faults.

However, to our knowledge, none of the previous work has addressed real-time aspects of embedded systems with executable assertions. In this work, we assess timing efficiency of assertions in order to reduce performance overheads of error detection and propose a

[1] We will refer to both intermittent and transient faults as *transient faults*.

framework for optimizing the assertion placing into time-constrained embedded systems. Our 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. [14]); and
4. select a set of assertions in terms of performance degradation and tightness (by using the optimization infrastructure).

Our approach can be useful for optimization of executable assertions in any embedded system with timing constraints. Examples of such systems include, but are not limited to, automotive electronics, airborne software, factory automation, and medical and telecommunication equipment. Our technique is useful not only for detection of transient faults but also for debugging of real-time programs with hard timing constraints. In such programs, assertions must not compromise the program's timing constrains, and, thus, have to be optimized.

The rest of the paper is organized as follows: the next section presents our application model, describes principles of error detection, and discusses basic properties of executable assertions; in Section III, we outline our problem formulation; in Section IV, we present our assertion placement optimization framework; finally, experimental results are presented in Section V.

II. Application Model

To represent application behavior, we have adapted the conditional process graph model proposed in [7] for the program instruction level. We represent an embedded program module M as a directed dependency graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}_S, \mathcal{E}_C\}$, where $\mathcal{V}$ is a set of nodes and $\mathcal{E}_S$ and $\mathcal{E}_C$ are sets of simple and conditional edges, respectively. The main difference from the model in [7] is that we will permit loops in the graph, i.e., the dependency graph $\mathcal{G}$ does *not* have to be acyclic, and we will consider program instructions instead of processes. Moreover, it is *not* required that the graph has to be *polar*, i.e., several source and sink nodes are possible. In our model, a node $I_i \in \mathcal{V}$ is a module instruction and e_{ij} is a direct dependency between instructions I_i and I_j, which can be, for example, a data or logical dependency. e_{ij} can be either a *simple* or *conditional* dependency. e_{ij} is a conditional dependency, i.e. $e_{ij} \in \mathcal{E}_C$, 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 e_{ij} is the only alternative for program execution, we will consider that it is a simple dependency, i.e., $e_{ij} \in \mathcal{E}_S$. Note that simple and conditional dependencies may constitute parts of module loops. The embedded program module M can be eventually implemented either is software (SW) or in hardware (HW) and will be referred as a *HW/SW module* or simply *module* in the paper.

In Figure 1 we present an example of a simple module M_1 and the corresponding dependency graph $\mathcal{G}_1$. This module calculates a factorial of an integer number N, where N is an input. Graph $\mathcal{G}_1$ consists of 6 instructions, I_1 to I_6, and 6 dependencies, e_{13} 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 e_{36} and e_{34} are conditional dependencies. e_{36} is taken if the "while" condition I_3 is "false" and e_{34} is taken if the "while" condition I_3 is "true". 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 $\mathcal{G}_1$ has two source nodes, I_1 and I_2, and one sink node I_6.

We will assign performance values to instructions and dependencies in the HW/SW module with the static/dynamic profiling of the module.[2] Performance values for module M_1 are shown in the bottom of Figure 1. For example, instruction I_4 will take 15 time units to execute. Time constraints, or *deadlines*, are assigned to the HW/SW modules. 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 Figure 1. Thus, instruction I_3 to I_5 are not allowed to execute more than 50 time units.

We will consider that transient faults may affect execution of program P_1 and are, thus, interested in optimization of assertion placement for the time-constrained module M_1.

```
Factorial(const int N)
I1: double factorial = 1;
I2: int i = 1;
I3: while(i <= N) {
I4:     factorial *= (double) i;
I5:     i++; }
I6: return factorial;
end Factorial
```

I_1	3	e_{13}	1
I_2	3	e_{23}	1
I_3	10	e_{34}	5
I_4	15	e_{45}	1
I_5	3	e_{36}	1
I_6	5	e_{53}	5

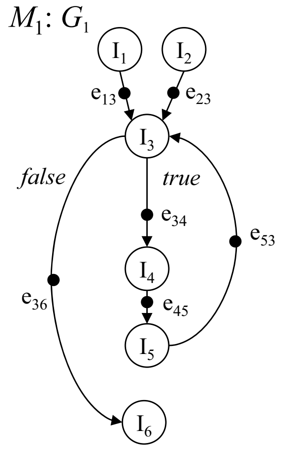

Figure 1. Example of a program and its instruction graph.

A. Error detection with assertions

Several errors can happen to the HW/SW module M_1 of program P_1 in Figure 1, 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 program 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 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". 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 deadline. Violation of this deadline may lead to catastrophic consequences, and, hence, must be avoided. 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 also lead to potentially catastrophic

[2] In general, these performance values are dependent on the actual execution sequence of the module. However, to simplify performance analysis, we will begin with the performance values individually profiled for instructions and dependencies, which will give us the first approximate performance figures of the module execution, as described in Section IV.A. Later, as described in Section IV.B, module execution sequences (with the introduced assertions) will be profiled to capture performance effect of inter-dependencies between instructions (and assertions) in these execution sequences, and, thus, deadline satisfaction will be ensured over all execution scenarios of the module.

consequences. Thus, both performance overheads of assertions and their efficiency have to be considered in the assertion placement.

B. Parameters of executable assertions

To capture effectiveness of assertions, we will 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 HW/SW module M against random faults 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 will compute them as a part of our profiling strategy described in Section IV.

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 [24] or with extensive simulations of program execution [25]. We will also compute them in the profiling step of the optimization framework, as described in Section IV.

In general assertions shall be introduced with the *highest possible tightness* at the *lowest performance degradation*.

In Figure 1, instruction I_4: `factorial*=(double)i` 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();>
```

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();>
```

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% for instruction I_4 (it captures faults only in the multiplication) and assertion A_2 gives tightness of 90%.[3] 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();>
```

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 the 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();>
```

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 Figure 1, given a list of assertions A_1, A_2 and A_3 and the deadline of 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. 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 Figure 1, decision on which assertion to choose is relatively straightforward, as the size of HW/SW module increases, these decisions becomes much more difficult. If in Figure 1 we have only three possible assertions, 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 a HW/SW module. For example, in Figure 2a, 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 will deliver its tightness only in 1 case out of 100. In 99 cases it will not contribute to the error detection of transient faults. We will consider that, if, for example, the initial *static* tightness of A_m is 80%, the actual *dynamic* tightness will be 80 / 100 = 0.8%. Thus, efficiency of assertions also depends a lot on how often they (and their related instructions) are executed. In Figure 2a, introduction of an assertion A_n with static tightness of 40% into the "false" branch will make more sense, i.e., its dynamic tightness will be $40 \times 99 / 100 = 39.6\%$, 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 Figure 2b, 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 will be obviously the best, despite the fact that its static tightness (40%) is twice as small as the static tightness (80%) of A_m.

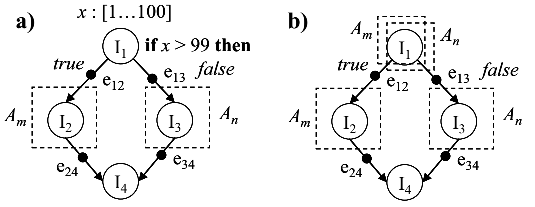

Figure 2. Example of a conditional execution of an assertion.

Thus, to address the complexity of assertion placement, we have proposed an assertion placement optimization framework described in Section IV, which solves a generalized problem of assertion placement that we present in Section III.

III. Problem Formulation

As an input we get a HW/SW module M (in VHDL, SystemC, C) of a time-constrained embedded system. Module M does not contain executable assertions. Several sets of instructions in this module M are associated with hard deadlines, as illustrated in the example in Figure 1. 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

[3] 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.

the module source code directly under the "_DEBUG" compilation flag.

As an output, we want to produce a HW/SW module with the subset of assertions introduced at the best possible places in the module source code, which maximize tightness, while meeting hard deadlines.

IV. Assertion Placement

In this section, we present the approach for placement, optimization, and evaluation of error detection primitives in time-constrained embedded systems. In particular, an overview of the developed framework is shown in Figure 3. 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.

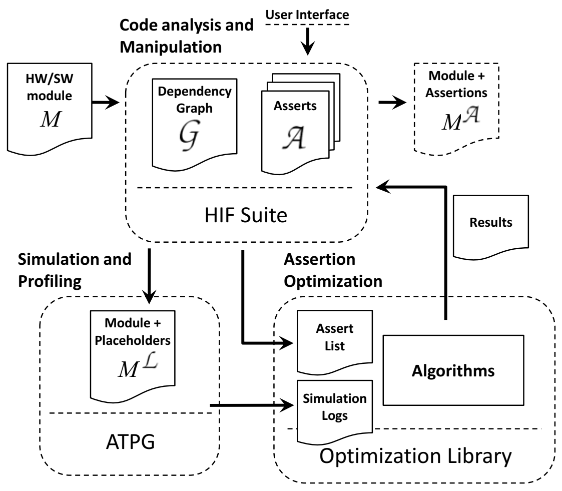

Figure 3. The profiling and optimization framework for assertion placement.

The framework takes as input an HW/SW module (VHDL, SystemC, C) 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 [3]. In this phase, the framework introduces assertions, i.e. $\mathcal{A}$, and placeholders by exploiting the dependency-graph associated with the module, i.e. $\mathcal{G}$. Each assertion is coupled with a statically computed value for performance degradation. Moreover, the framework allows the user to provide an actual assertion for each placeholder. Then, the module description with placeholders, i.e. $M^{\mathcal{L}}$, is simulated for generating further 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 can be easily integrated. The generated profiling information is a simulation log which, for each placeholder, contains the dynamically computed 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 system, and optimized in terms of error detection.

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

A. Code analisys and profiling

The analysis and manipulation of the module descriptions are based on HIFSuite [3], 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. 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 $\mathcal{G}$, 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 Figure 4.

```
if (x <= max) {
  // then body
  <if (!(x <= max)) error();>
  x = y + z;
  // then body
} else {
  // else body
  <if (x <= max) error();>
  // else body
}
```

Figure 4. Protecting conditional-statement branches against soft errors.

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 Figure 5, 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 further assertions or modify the assertions that the framework automatically choose.

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

Figure 5. Protecting array reading against soft errors.

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, further values for tightness and performance degradation are dynamically-computed and registered. An intuitive example of tightness computation for executable assertions is reported in Figure 2. 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 in terms of time units.

B. Optimization infrastructure

In the previous phases, a set of candidate assertions addressing soft errors is associated with each module of the time-constrained embedded system, and simulation-based profiling information is generated. These assertions have different probability of detecting errors, and increase 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 system.

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).
- An *assertion-dependency graph, ADG = {$\mathcal{A}$, E^A}*, where each node is a candidate assertion ($A_m \in \mathcal{A}$) and each edge (i,j) $\in E^A$ correlates two assertions, if assertion *j* is the subsequent of assertion *i* during the simulation.
- 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 *Best Assertion First* (BAF) strategy, shown in Algorithm 1. It always chooses, while the deadlines are respected, the candidate assertion with the *highest probability* of detecting errors. In such a case, we assume that the error detection probability is proportional to the static performance degradation 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, the set of candidate assertions, and a maximum-tolerated-overhead value, which depends on the time constraints of the embedded system. It returns a set of assertions (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 (line 6). Then, the assertions are selected till they exhaust the available tolerated overhead (lines 7-13).

```
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 ← perf_deg_order (CAS);
7.  while remaining_overhead ≥ 0 ∧ flag 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
```

Algorithm 1. The Best Assertion First (BAF) algorithm.

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 static 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.

C. Evaluation metric

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, $\mathcal{A} = \{A_1, ..., A_K\}$ a list of candidate assertions for the module *M*, and p an assertion-placing algorithm. Then the result of p over *M* and $\mathcal{A}$ is the set of assertions $\mathcal{A}_p = \{A^p_1, \dots, A^p_H\} \subseteq A_M$. The quality of $\mathcal{A}_p$ is measured as the ratio

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

where τ_j and δ_j are, respectively, the tightness and performance degradation associated with each assertion A_j in $\mathcal{A}$, while τ_i^p and δ_i^p are associated with each assertion A_i in $\mathcal{A}_p$.

Thus, this metric not only considers the quality of the selected assertion, but also takes into account the frequency of assertion execution (the dynamically computed tightness).

V. Experimental Results

In order to assess the effectiveness of the proposed framework, we have used the benchmarks described in Table 1, where columns Pi, Po, and Var respectively report 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. Such benchmarks are from ITC'99 suite, that is a well know reference used by other authors [5].

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

Table 1. Benchmarks characteristics.

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 IV.C.

In Table 2 and Table 3, the results of the algorithms are reported in columns Baf, Faf, and Meaf for benchmarks *b03* and *b05*, respectively. We have chosen to report only the results of *b03* and *b05* for lack of space and because these benchmarks are the biggest in terms of candidate assertions. We executed each algorithm with a maximum tolerated overhead, which is reported in column Mto. Such a value is expressed as a percentage on the overall overhead (column Oh in Table 1), i.e. 5%, 10%, … , and 25%. For example, in the case of *b05* the Mto values are 5ns, 10ns, 15ns, 20ns, and 25ns. 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 Table 1.

	Baf		Faf		Meaf	
Mto (%)	Ratio	Sas	Ratio	Sas	Ratio	Sas
5	0.125	3	0.026	7	0.290	5
10	0.246	6	0.272	13	0.415	8
15	0.360	9	0.396	16	0.540	11
20	0.383	8	0.518	18	0.665	11
25	0.455	12	0.632	20	0.791	13

Table 2. Comparison of the optimization algorithms (b03 benchmark).

	Baf		Faf		Meaf	
Mto (%)	Ratio	Sas	Ratio	Sas	Ratio	Sas
5	0.182	8	0.027	16	0.289	10
10	0.192	14	0.075	34	0.426	18
15	0.295	21	0.125	51	0.557	28
20	0.407	50	0.317	67	0.637	41
25	0.448	36	0.359	80	0.704	56

Table 3. Comparison of the optimization algorithms (b05 benchmark).