Adjustable self-healing methodology for accelerated functions in heterogeneous systems

Mohammad Riazati*, Tara Ghasempouri†, Masoud Daneshtalab*, Jaan Raik†, Mikael Sjödin*, and Björn Lisper*
*Heterogeneous Systems Research Group, Mälardalen University, Västerås, Sweden
†Department of Computer Systems, Tallinn University of Technology, Tallinn, Estonia
*{mohammad.riazati, masoud.daneshtalab, mikael.sjodin, bjorn.lisper}@mdh.se
†{jaan.raik, tara.ghasempouri}@taltech.ee

***Abstract*—Self-healing is a promising approach for designing reliable digital systems. It refers to the ability of a system to detect faults and automatically fixing them to avoid total failure. With the development of digital systems, heterogeneous systems, in which some parts of the system are executed on the programmable logic, and some other parts run on the processing elements (CPU), are becoming more prevalent. In this work, we propose an adjustable self-healing method that is applicable to heterogeneous systems with accelerated functions and enables the designers to add the self-healing feature to the design. In this method, by manipulating the software codes that are being executed on the processing element, we add the ability to verify the accelerated functions on the programmable logic and heal the possible failures to the system. This is done not only in a straightforward manner but also without being forced to choose a specific reliability-overhead point. The designer will have the option to select the optimum configuration for a desired reliability level. Experimental results on a large design including several accelerated functions are provided and show 42% improvement of reliability by having 27% overhead, as an example of the reliability-overhead point.**

***Index Terms*—Self-healing, Acceleration, Heterogeneous systems, Reliability, Genetic algorithm**

I. Introduction

Heterogeneous system design, also known as Hardware/Software (HW/SW) co-design, is a design paradigm introduced in the early nineties and has been studied extensively by the system design community. Tremendous progress has been achieved in this area of research. This has brought new thinking to many aspects of computer system design, including process scheduling, communication protocols, memory management, software development, as well as the design of application-specific processors and re-configurable architectures [1], [2].

The failure in the hardware, which is one side of the heterogeneous system, is one of the areas covered by the researchers. They may occur in a system because of the aging of the hardware or the impacts from the surrounding environment (e.g., radiation, temperature, etc.) [3]–[5].

Besides the works on the failures in the hardware, self-healing is also an addressed technique in the area of large and complex systems [6]. Self-healing tries to keep the systems alive and running despite the possible failures. This is vital for many real-life systems such as hospital facilities, aircraft, etc. Regardless of the domain these systems belong to, all of them may face failure in any part, which may reduce or hamper their performance [7].

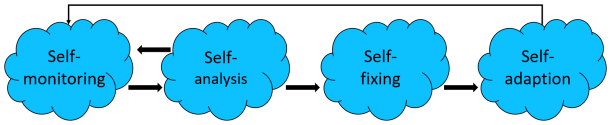

Fig. 1. Generic loop of a self-healing system

To handle such failures, self-healing has been envisioned as a solution with a promise to keeping the system available and functioning all the time. By self-healing, the system's components are expected to solve or repair some of or all possible damages without requiring external intervention.

Generally, the loop of self-healing for any system can be divided into four main phases [7]–[10]. Figure 1 demonstrates the generic flow of a self-healing system. *Self-monitoring* monitors the system to inspect the environment to collect the information required to detect the failure. It will send the data gathered through current observations to the next stage. *Self-analysis* analyzes the system according to the predefined requirements and expectations. If the analysis shows that everything is according to expectations and requirements, the flow goes back to its initial state, self-monitoring. Otherwise, the detected error will be reported to the next stage of the cycle. In *Self-fixing*, the error is analyzed, and a method or a strategy of repairing is determined and performed. The results of the problem correction operations are reported to the final phase. *Self-adaption* is responsible for making sure that the recent correction will not cause a problem to the overall system function as well as individual components. This might include informing other components or the system controller. Once the faulty areas are self-healed, the cycle begins all over again. This cycle continues forever until the time the system is shut down or the healing is not possible anymore.

For pure hardware systems, some methods for self-healing are proposed. They include Dual Modular Redundancy (DMR) [11], Triple Modular Redundancy (TMR) [12], Embryonic Hardware (EmHW) [13], [14], artificial hormone system [15], [16], Evolvable Hardware (EHW) [17] and intelligent frameworks that are designed with healing capabilities in mind [18].

DMR and TMR are two types of redundancy techniques. In the DMR technique, two identical instances run in parallel for the same function, and their outputs are connected to a comparator that sends a mismatch indicator whenever a fault occurs. The drawback of this method is that there are two active components and a voter part for fault detection,

Preprint submitted to Euromicro Conference on Digital System Design (DSD'20)

which obviously brings a significant area overhead. TMR is a relatively similar approach with three identical modules, and all of them are active simultaneously and execute in parallel to provide fault masking through majority voting of their outputs to mask a signal fault. In case any arbitrary function module becomes faulty, the whole system will continue working correctly as the TMR system, as a result of the majority voting mechanism, discards the output of the faulty module and directs the output of the correctly working module to the output of the whole system. Clearly, the drawback of this method, too, is the high redundancy costs. EmHW is a hardware system of self-healing based on the replication of small building blocks with identical architectures, and it has been inspired by the multi-cellular organisms' cell division and differentiation mechanisms. Similar to multi-cellular organisms in nature, the structure of circuits based on EmHW is a two-dimensional array of electronic cells (e-cell) with a hardware reconfiguration platform. When a cell fails, the responsible module triggers self-healing, and the faulty cell will be replaced by the spare cell. However, the drawback of such techniques is their demand for significant additional hardware that leads to extra area and power overhead. EHW is another bio-inspired self-healing system. EHW is a reconfigurable hardware system that evolves by the control of some Evolutionary Algorithms (EAs) to solve a real-time task. EHW can be used in a self-adaptive platform that adapts to its working environment without external human intervention. The evolutionary algorithm is implemented to provide a self-evaluation of solution candidates and choose the best candidate by tracking the cost function. With such a realization, it is able to recover and survive against degradation and faults. It also enhances the performance continuously during the whole lifetime of the operation.

There are also some self-healing approaches in the software domain. They cannot be used as a self-healing method for the accelerated modules. Nevertheless, they can be applied to the software part of the heterogeneous systems. SHADOWS (Self-healing Approach to Designing Complex Software Systems) [19] proposes an automatic detection and repair of possible problematic behavior in its early design and development stages. PANACEA framework [20] is another approach based on the SHADOWS objects that provides a design methodology as well as ready-to-use healing elements aimed at enhancing software systems with self-healing capabilities both at design time and at run time.

Although the approaches proposed for the hardware are innovative and relatively effective, they suffer from shortcomings such as high area-overhead and high cost. Moreover, regarding the reliability level of each of these proposed methods, if the designer selects one of them, they should accept the proposed reliability-overhead point, even if the acquired reliability is more than their requirements. Thus, to fill in this gap and overcome the current shortcomings, this work proposes a method to add the self-healing feature to a heterogeneous system with an adjustable reliability-latency point and minimal area overhead. Furthermore, since the self-healing method is implemented through manipulation of the code that is running on the existing processing element of the heterogeneous system, the area overhead on the FPGA fabric is zero.

The input to our method is a heterogeneous system that both software and hardware versions of its accelerated functions are available. Indeed, we exploit the fact that an accelerated function might be generated from a software code in a high-level language through a high-level synthesizer (HLS). Availability of both versions enables us to use the software version to verify the correct functioning of the hardware version and to use it as a substitution for the hardware version. However, the difference between the latencies of these two versions might devastate the efficiency of the whole system, and thus, we have proposed an adjustable and optimized solution to make it practical in real applications.

Our contributions in this paper can be summarized as follows:

- A self-healing method is proposed that is applicable to the heterogeneous systems with accelerated functions to prevent the failures of the system.
- An adjustable self-healing feature is provided that gives the ability to the designers to set the desired reliability point based on the latency requirements.
- The proposed method is applicable to heterogeneous systems regardless of their complexity without any hardware overhead.
- A set of Pareto-optimal configurations with regard to the latency overhead and desired reliability level is formed.

The rest of the paper is organized as follows. Section II introduces preliminaries related to this work. Section III presents the methodology. Section IV describes the experimental results. Section V discusses the observations during the implementation and experiment phases, and finally, Section VI concludes the paper.

II. Preliminaries

As the focus of this work is on heterogeneous systems and accelerated functions, in this section, the necessary preliminaries on these topics are provided.

In the past, when designers were to start the implementation of a specific system, given that the system was implementable both as a software running on the processor or as a hardware running on an ASIC or FPGA chip, they had to make the decision based on the performance of the whole design on each of these platforms. However, nowadays, with the advent of the heterogeneous integrated circuits (ICs), the flow has changed. Recently, Intel provides SoC FPGAs such as Agilex and Statix 10, and Xilinx provides various families of Zynq products. Both products integrate multiple processing elements together with the FPGA fabric. As said, in the past, the designers had to choose either the FPGA or the processor to run a particular program while these heterogeneous chips allow the designer to split the design into two parts and exploit the advantages of both platforms. For each segment of the design, e.g, each function, the designer assesses its performance on hardware and software, and after considering the costs, finally decides

on the preferred location to put the design and run it. This scenario is also known as function acceleration.

Hardware (HW) acceleration can make a computation several times faster than running the application on processors, due to their ability of massive parallel execution. That is, HW accelerating aims at decreasing the latency and increasing the throughput in computational tasks. It is noteworthy that this efficiency is usually achieved in computation-intensive functions. For memory- or data-intensive tasks, this might not be the case as the access to the memory is considered as the bottleneck and prevents the programmable logic from exploiting the possible parallelism.

The accelerator might be designed as a hardware module from scratch. However, as most of the time, the design is already available in a high-level language, the designer may opt to automatically transform the existing design to the hardware. This process is called High-Level Synthesis (HLS) and usually takes much less time than designing the accelerator from scratch and consequently has become prevalent. Most of the HLS tools, including Intel HLS Compiler and Xilinx Vivado HLS, receive the C/C++ as the input. In this paper, we have focused on the C/C++ syntax. However, the method is applicable to other languages too.

Functions are considered as one of the most distinguishable segments of a design. As a result, most of the tools for designing accelerators on SoC FPGAs, such as Xilinx SDSoC tool, which we use to collect the experimental results in this paper, consider the functions as the smallest program segments to be accelerated. This is not basically considered as a limitation because if the designer wants to accelerate a piece of code that is not a function, can encapsulate that segment into a function.

In heterogeneous systems that exploit the acceleration of the functions, the program starts running on the processor, and when it reaches a function call to an accelerated function, the software execution flow halts, and the execution continues on the hardware. After the execution is over on the hardware, the result is passed to the software through some communication facilities like registers or the Direct Memory Access (DMA). Then, the program continues running on the processor. In this work, we assume the design, as shown in Fig. 2. The upper part of the image shows the original design, which may run on the processor. The lower part, instead, is the accelerated version of a hypothetical design in which some functions are accelerated. The parts of the code specified as Seg., are the code segments not designated to be accelerated, and the parts specified as Func. are the function calls to the accelerated functions. The assumption is that the accelerated function runs faster than its corresponding software version, and in this paper, we do not go through the acceleration efficiency and necessity, and the focus is on adding the self-healing feature to existing accelerations.

III. Proposed Method

In this paper, we present a method to help the designers easily integrate the self-healing feature into a heterogeneous system. The solution is generic and applicable to all HW

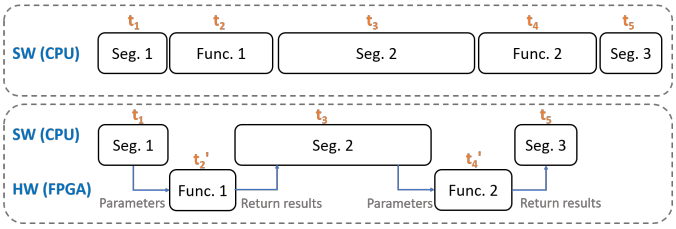

Fig. 2. Acceleration of the functions

accelerators where both hardware and software versions of accelerated functions are available; the existence of both versions is normally the case when the accelerators are made by the HLS process. Unlike most of the reliability methods, our solution has no area overhead on the FPGA fabric, but rather would degrade the performance. The main latency overhead is in the case of a failure, which is the cost of keeping the system alive. The latency overhead during the normal operation for evaluation is decided and determined by the designer. The reliability performance is adjustable, even at run time. Based on the desired reliability, the self-healing parameters are adjusted in a way that the minimum latency overhead is incurred. Additionally, if a design consists of more than one accelerated function, a Pareto-frontier, including all the required parameters, is provided to simplify the decision making by the designer.

In the rest of this section, two main parts of the methodology are described. Firstly, Subsection III-A introduces how self-healing features should be added to the design. This includes the implementation of four standard self-healing phases, the required modifications on the design's source, and the description on what parameters and variables are needed. The values for these parameters directly affect the reliability level and the overall latency of the system. Secondly, Subsection III-B proposes a multi-objective genetic-algorithm-based optimization method to find all the Pareto-optimal reliability-latency points.

A. Self-healing framework

The proposed method needs a heterogeneous system as an input, as explained in Section II, and it is assumed that for the accelerated functions, both software and hardware versions are available. This assumption can help the designers to evaluate the correct execution on the hardware by also running the software instance.

Before describing the proposed methodology in detail, some frequently used abbreviations are presented here to make the naming brief and ease the process of reading:

- **ACCF**: the accelerated function. By calling this function, the execution on the processor halts, and this function executes on the FPGA.
- **SWF**: the software version of the function. This function runs on the processing element of the heterogeneous system.
- **HLF**: the self-healing function, which calls ACCF and SWF. The core functionality of the self-healing is implemented in this function.

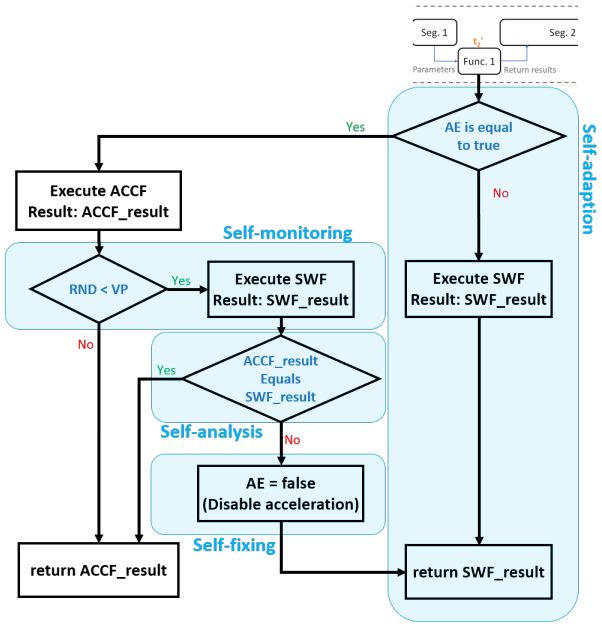

Fig. 3. The proposed self-healing framework

- **AE**: is a global variable indicating whether the accelerated version is enabled or not. The HLF is responsible for toggling this variable. Its initial value is true, which means the hardware version (ACCF) is not faulty and is working correctly.
- **VP**: is the Validation Probability of a function. It determines how frequently the SWF is called to assess the correct execution of the ACCF. The higher the VP is, the higher the reliability is, the higher the average execution time of the whole system is, and the lower performance the system has.
- **RND**: is a randomly generated value between zero and one. This, along with the VP, determines the random execution of the SWF. If RND is less than VP, the SWF is executed.

Figure 3 shows an overview of the proposed self-healing framework. The input of the framework is each of the accelerated functions in the heterogeneous system.

For each function, a global variable is defined as AE. This variable determines whether the accelerated version of the function is active (not faulty) or not. Initially, this variable is equal to true, which means the accelerated function is enabled and working as expected. In the beginning, ACCF is executed, and the result is stored as ACCF_result. Then, the self-monitoring phase begins.

As demonstrated in Figure 3, in the self-monitoring phase, a random number (RND) is generated. RND would get a value between zero and one. If this value is less than VP, SWF is executed. This will result in a random validation with a probability of VP. If RND is greater than VP, the self-monitoring phase is ended, and the result of the ACCF, which is ACCF_result is returned as the final result. In contrast, if RND is less than VP, the SFW is executed too. In this case, the self-analysis phase will follow.

The values of the VP (the probability of running the function also on the processor) for each of the accelerated function have a major impact on both the reliability of the system and its overall latency. The higher this probability is, the longer the average execution time will be. In turn, when the validation through running the software takes place more often, the design becomes more reliable (redundant execution). For a design with only one function, this problem is a simple one-variable problem and can easily be solved by the designer. However, for a more complex design that each function has various execution time and vulnerability level, finding an appropriate probability for each function will become more complicated. We will propose an optimization method to find the Pareto-optimum points in section III-B.

In the self-analysis phase, the results of the ACCF and SWF are compared. If they are equal, it means that the ACCF is working properly, and the result of the ACCF, or the result of SWF as well, will be returned as the final result. Note that in this work, we only consider the possibility of fault in the accelerated function. We will discuss this assumption in the discussion section. If the results of the two executions are unequal, then the self-fixing phase will be necessary.

In the self-fixing phase, the acceleration is disabled by setting AE to false, and the result of the software version is returned. This will affect the future executions which are handled in the self-adaption phase.

In the self-adaption phase, having the value of false for AE indicates that the acceleration is disabled. In this case, only the software version is executed. Note that the execution of the SWF is most probably longer than the ACCF. This higher latency is the cost of keeping the system alive and correct instead of letting it malfunction.

Algorithm 1 shows how this flowchart should be implemented in the source C/C++ code. The HLF is responsible for managing the whole self-healing flow. The only remaining thing which should be done by the designer is to look for all the function calls to the ACCF in the design and replace it with the calls to HLF.

B. Finding the Pareto-optimal validation probabilities

In the previous section, we discussed the framework of the proposed self-healing mechanism. Most of the implementations are either already available from the design phase or implemented according to the proposed algorithm and self-healing function. Nonetheless, the value of the VP (Validation Probability), hence the frequency of validation, is still unknown and should be determined by the designer. As explained earlier, this value directly affects the performance of the whole design, which is a design tradeoff between reliability and performance. Based on the relation between the VP and overall latency of the design, the designer should decide on this parameter. For a design with only one accelerated function, this is fairly simple. Figure 4 depicts the effects of the various VP values on the overall latency of the CHStone AES benchmark [21]. The two horizontal lines on the graph show the latencies of the accelerated function (ACCF) running

Algorithm 1: Healing Function (HLF) implementation

```
AE = true;
function SWF (arguments) {
    Some processing on the input arguments;
    return result ;
}
function ACCF (arguments) {
    Some processing on the input arguments;
    return result ;
}
function HLF (arguments) {
    if AE then
        hw_result = ACCF(arguments);
        if RND < VP then
            sw_result = SWF(arguments);
            if hw_result ≠ sw_result then
                AE = false;
                return sw_result
            else
                return hw_result
            end
        end
        return hw_result
    else
        sw_result = SWF(arguments);
        return sw_result
    end
}
```

on the FPGA and its software version (SWF) running on the processor: 34,872 and 57,120 clock cycles respectively. The experimental setup to obtain these values is explained in Section IV. The graph shows that if VP is equal to 0.1, which means that 10 percent of the times the validation of the ACCF occurs, the overall latency of the AES function will increase by 16 percent, from 34,872 to 40,584. This latency overhead is the cost of the overall reliability of the system and keeping it working faultlessly though at a higher latency. This is up to the designer to find the appropriate VP based on the design requirements. The overall latency overhead is calculated based on Relation 1. In this relation, $L(F_x)$ is the latency of the function F_x.

$$\frac{(1-VP)\times L(ACCF)+VP\times\Big(L(ACCF)+L(SWF)\Big)}{L(ACCF)} \quad (1)$$

As explained, the relation between VP and the overall latency for a single-function design is a linear single-variable function. Hence, finding the desired reliability level is simple for the designer. However, for a multi-function design, it will be more complex. In order to simplify finding an appropriate VP for each function, we have developed a genetic-algorithm-based algorithm to find the Pareto-optimal points. To begin with, we define some symbols, some of which are determined by the designer and some others by the proposed algorithm.

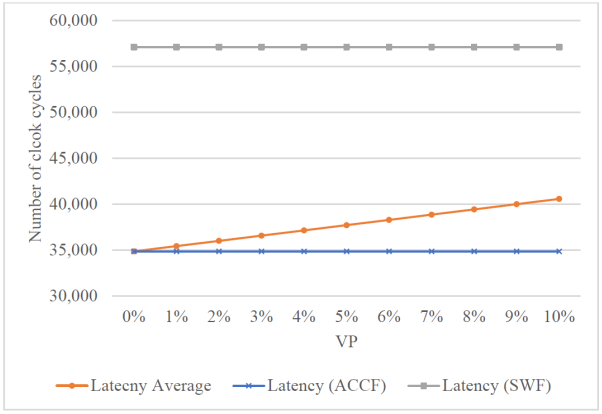

Fig. 4. VP effects on latency in a single-accelerated-function design

- **LACC**$_i$: is the Latency of the ith Accelerated function. They can be either the number of clock cycles or the wall-clock time. The former is recommended as it makes it independent of the chip's clock frequency.
- **LSW**$_i$: is the Latency of the SoftWare version of the ith function. The type of this latency should match LACC's.
- **VL**$_i$: is a comparative parameter indicating how a function contributes to the total vulnerability of the whole design. By setting VL$_i$ greater than VL$_j$, the designer will inform the proposed algorithm that the ith function is either more vulnerable to failure or is more important to be working correctly. Generally speaking, the higher value for a VL will result in a higher frequency of validation (VP) and a larger latency resulted from a specific function. If the designer considers all the functions of equal importance and vulnerability, they can assign equal values to all VLs.
- **VP**$_i$: as explained earlier, is the validation probability of the ith function. The values of VP are the outputs of our proposed optimization algorithm. Based on the other parameters provided by the designer, for each member of a set of possible desired overall latencies, a set of VPs are provided for the designer, which are the pareto-optimal points of Overall Vulnerability Degrees versus total execution time.
- **OVD**: this is the expected Overall Vulnerability Degree of the whole system. This is a function of the VPs and VLs as shown in relation 2. If all the VPs are equal to zero, the OVD will be equal to one which corresponds to the higher vulnerability degree. On the other side of the spectrum, if all the VPs are equal to one, which means the SWF is always executed (which is not usually a desired point) gives the minimum vulnerability of zero. As explained in the previous item, for each of the possible values for overall vulnerability degree, the optimum set of VPs are suggested, and accordingly the overall latency is calculated.

$$OVD=\frac{\sum(1-VP_i)\times VL_i}{\sum VL_i} \quad (2)$$

Based on the above parameters, two objectives are considered. The first objective is to minimize the overall vulnerability level and the second objective is to minimize the total execution time, which is elaborated in relation 3. The execution time for each function is the probability of running the ACCF times the latency of the ACCF, plus the probability of running both ACCF and SWF times the total of their latencies.

$$\sum(1 - VP_i) \times LACC_i + VP_i \times (LACC_i + LSW_i) \quad (3)$$

C. Multi-objective genetic algorithm

Since the search space for all possible configuration was very large, we were motivated to use the Genetic Algorithm (GA) to find the Pareto-optimal VPs.

In order to solve the optimization problem, we implemented a multi-objective genetic algorithm (MOGA) [22] in C++. Despite the single-objective GA, which is based on the fitness functions and their optimum values, MOGA utilizes the notion of dominance and tries to form a population in which the individuals have the minimum number of dominating points in the solution space. Other concepts, such as mutation, crossover, and elitism are analogous in both methods.

In genetic algorithm, the inputs are grouped together as the chromosomes. In our case, each chromosome consists of all the VPs. The fitness functions are calculated according to relation 2 and 3. The input to the MOGA are three constant vectors LACC, LSW, and VL. In this specific implementation, we did not consider any specific constraints for the solutions except that the values of the VPs must stay in the valid range of probabilities, which is a decimal number between zero and one. The algorithm begins with an initial population of individuals created randomly. In the elitism phase, we selected ten percent of the individuals from the previous generation that had the minimum number of dominating individuals. After sorting the individuals in a population in an increasing order of the number of dominating solutions, an offspring is the result of mating two parents from the first half of the population. The mutation rate in the mating phase was considered as twenty percent.

IV. Experimental results

In order to estimate the efficiency of the proposed method, we applied it to a five-function instance. The used parameters are listed in Table I. The first column presents the id of five function calls in the experiment. As described, LACC is the latency of the accelerated function. 482, 280, 107, 374 and 166 clock cycles are reported for the latency of each accelerated function call, respectively. LSW column describes the latency of the software version of the functions. 2096, 962, 346, 1230, and 695 clock cycles are reported for the latency of the software functions, respectively. To set the values for the VLs, we assumed that the functions with a longer execution time are more important and more vulnerable. However, this assumption might not always be a correct one. We have used these parameter values, instead of setting all the VLs to an equal value, to show the capability of the proposed method to consider the differences in the vulnerability and the importance of the functions; the designer is expected to set the values of the VLs according to the assessed vulnerability and importance of each function.

TABLE I
The input parameters of the optimization algorithm

Function Id	LACC	LSW	VL
1	482	2096	0.2
2	280	962	0.3
3	107	346	0.9
4	374	1230	0.2
5	166	695	0.6

Figure 5 shows the result of running the two-objective optimization for 40 epochs. The population of each generation was 250 individuals. Each of the dots in this figure corresponds to a 5-tuple of VPs. For instance, the dot on the top left corner of the graph is when all the VPs are set to zero, which means that no validation occurs, and only the accelerated functions are executed. As shown, the total execution time at this point is the sum of the values in the LACC column in Table I. On the other end of the spectrum, the point on the bottom right corner of the graph is when the LSW is always executed along with the ACCF, which is the result of having the value of 1.0 for all the VPs. The curve in the graph is the pareto frontier of the points, which means that by picking a desired OVD, what the minimum possible total execution time will be. The rest of the points are Pareto dominated by this orange curve.

Table II shows the corresponding values of the VPs for some of the points on the Pareto-frontier curve. The values in the VP_i columns are the validation probabilities found through the optimization algorithm. The Exec. Time column presents the total Execution Time and the Vuln. Degree represents the overall Vulnerability Level, which are the two objective of the optimization algorithm and they are calculated using relations 2 and 3. As can be seen, the minimum execution time and the maximum vulnerability correspond to setting all the VPs to 0, which is the case that the healing is disabled for all the functions of the design. The other end of the spectrum is when

Fig. 5. The Pareto frontier of the various VP settings and their effect on the OVD and the total execution time

TABLE II
CORRESPONDING VPS FOR SOME OF THE POINTS ON THE PARETO FRONTIER OF FIGURE 5

Exec. Time	Vuln. Degree	VP_1	VP_2	VP_3	VP_4	VP_5
1409	1	0	0	0	0	0
1795	0.5845	0	0	0.98	0.01	0.05
2236	0.4340	0.01	0.07	1	0.02	0.53
2396	0.3536	0	0.06	1	0	0.84
2707	0.2863	0.01	0.24	1	0.01	0.99
3428	0.1840	0.01	0.99	1	0.01	0.99
4053	0.1427	0.04	0.98	1	0.48	0.98
4597	0.0977	0.01	0.99	1	0.96	0.99
6109	0.0431	0.87	0.99	1	0.73	0.98
6738	0	1	1	1	1	1

all the VPs are set to 1, which means that the validation takes place on every single execution of the function. In this case, the OVD is the minimum, and the execution time is the maximum, which is about five times slower than the original design's latency. As explained before, the designers are expected to select one set of the VPs that corresponds to their desired reliability.

V. DISCUSSION

We performed several studies to choose the best approach for the proposed methodology. All the pros and cons of these alternative approaches were investigated, and after a careful analysis of different aspects, the proposed methodology was defined, implemented, and experimented. In the following, a brief explanation about these alternative approaches and the reasons which led us to present the proposed methodology are reported. Note that from the optimization point of view, all these approaches are the same, and the only change will be in the objective function calculation in relation 3. However, regarding the detection and healing flow, some fundamental changes are required.

In this work, we proposed running the software version of the accelerated function in addition to the hardware to compare the results and detect possible failures. They run in a sequential manner. Some alternate approaches can be considered that run both versions in parallel. Such parallelism can be implemented in several ways that we list here.

- Running the accelerated version and the software version in parallel: running both versions in parallel will certainly improve the performance by decreasing the overall latency. However, this parallelism cannot be implemented via the standard heterogeneous system design tools such as Xilinx SDSoC. In these tools, when a function is selected for the acceleration, the tool will take care of the necessary communications and handshaking between the processing system (PS) and the programmable logic (PL). They implement the procedure of the program control taking-over in a way that the main body of the program halts until the results of the accelerated function are ready and sent back to the PS. In order to implement such parallelism between software and hardware, the connections and the program control take-over procedure should be implemented manually.

TABLE III
COMPARISON BETWEEN ALTERNATIVE IMPLEMENTATIONS

Method	PL/PS Comm.	Speedup	Late detec.
Proposed method	Handled by the design tool	Baseline	No
Parallel exe. blocking	Custom. implementation	Minor	No
Parallel exe. non-blocking	Custom. implementation	Major	Yes

- Blocking or non-blocking approaches: given that the parallelism is implemented, two approaches can also be considered for the self-analysis phase. The execution of the rest of the program, which is after the current accelerated function, can either halt or continue; the rest of the program can be both the normal parts of the program or the next accelerated function. If the execution halts, the extra efficiency obtainable by parallelism would be limited, and if the execution continues, the late failure detection might occur. Late failure detection means that the rest of the program uses the results of a faulty accelerated function, and after executing some statements or the next accelerated function, the fault is realized, which might be too late for a reaction.

In Table III, the comparison of these alternatives is provided in brief.

We also had several observations during the experiment phase of the proposed methodology in the real-life problems, which are addressed as follows:

- By running the method on various sets of functions, we realized that the use of this method might be limited when the performance of the accelerated function is much higher than the software version. For instance, to exaggerate the issue, consider a function that its accelerated version is a thousand times faster than the software version. In this case, even if the VP is set to 0.1 percent, the overall performance is degraded to almost half. This fact should be considered by the designers, and they should pay this cost only if the reliability of the target function is worth it.
- One of the challenges of using the proposed method is the possibility of failing the software. First of all, there are several research results on self-healing software such as [23]–[25], and some others that were introduced in section I that can be used along with our proposed method. Besides, we have also considered an extension of this method to make it more reliable. The idea is that we can choose a fraction of the function's input combinations, run them offline, and keep them in the memory. During the heterogeneous self-healing, we can also compare the results with this set of pre-computed results as an extra source of evaluation. This is a challenging problem with regard to finding the most fault-covering pre-evaluated inputs while considering the memory overheads incurred. We may address this idea in our future works.
- As this self-healing feature increases the latency of the system, its application to the real-time systems is limited. There are a few scenarios that a real-time system can use this method. One of them is the situations that the real-time system, for some reason, has no task to do, and it

can use our method in this free time just to verify the accelerated functions. However, even in this case, it is apparent that if a failure is detected, the software cannot replace the accelerated function, due to the excessive latency of the software counterpart.

- In this paper, we did not cover the possibility of soft errors, which disappear after some time. Nevertheless, this feature can be simply added to the proposed method. As stated earlier, AE indicates if a specific accelerator is disabled due to an error detection. In order to consider the possibility of a soft error, this variable should be reset once in a while. The designers should note that the frequency of this AE resetting will affect the overall system performance.
- The healing flow is managed and implemented in the software part of the proposed method. Consequently, this enabled the designer to adjust the reliability level even at run time. The designer can even completely disable this feature at run time if needed. We have not covered this dynamic adjustment method as it needs the designer's knowledge on how this dynamism should behave. However, it can be implemented as a separate add-on module to our method that generates appropriate VPs dynamically.

VI. Conclusions

In this paper, we started by introducing a self-healing method for the heterogeneous systems. Subsequently, we provided the implementation guide of the method, and presented an optimization method to adjust the required parameters such as reliability performance.

Based on the desired reliability, the self-healing parameters are adjusted in a way that the minimum latency overhead is incurred. Furthermore, the method proposes an adjustable reliability based on the desired performance, even at run time. Additionally, if a design consists of more than one accelerated function, a Pareto-frontier of all the possible configurations is provided to simplify the decision making by the designer.

In the experimental results, we examined the method on a fairly large system and illustrated the results. We showed how much the reliability of the system is upgraded by using this method. Unlike most of the reliability methods, our solution has no area overhead on the FPGA fabric. The main latency overhead is in the case of a failure, which is the cost of keeping the system alive.

References

[1] X. S. Hu, R. C. Murphy, S. Dosanjh, K. Olukotun, and S. Poole, "Hardware/software co-design for high performance computing: Challenges and opportunities," in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2010, pp. 63–64.

[2] V. Thangavel, "Cascaded digital refinement for intrinsic evolvable hardware," 2015.

[3] A. Maiti, L. McDougall, and P. Schaumont, "The impact of aging on an fpga-based physical unclonable function," in *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, 2011, pp. 151–156.

[4] S. Srinivasan, P. Mangalagiri, Y. Xie, N. Viiaykrishnan, and K. Sarpatwari, "Flaw: Fpga lifetime awareness," in *2006 43rd ACM/IEEE Design Automation Conference*. IEEE, 2006, pp. 630–635.

[5] J. Cheng, C. Zhou, D. Yu, Z. Qian, and M. Chai, "Characteristic analysis and modeling for the near field radiation of fpga," in *2017 IEEE 5th International Symposium on Electromagnetic Compatibility (EMC-Beijing)*. IEEE, 2017, pp. 1–4.

[6] K. Khalil, O. K. Eldash, and M. Bayoumi, "A novel approach towards less area overhead self-healing hardware systems," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2017, pp. 1585–1588.

[7] D. Ghosh, R. Sharman, H. R. Rao, and S. Upadhyaya, "Self-healing systems—survey and synthesis," *Decision support systems*, vol. 42, no. 4, pp. 2164–2185, 2007.

[8] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the First Workshop on Self-Healing Systems*, ser. WOSS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 27–32. [Online]. Available: https://doi.org/10.1145/582128.582134

[9] K. Kasem, E. Omar, K. Ashok, and B. Magdy, "Self-healing hardware systems: A review," *Microelectronics Journal*, 2019.

[10] K. P. Kumar and N. S. Naik, "Self-healing model for software application," in *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*. IEEE, 2014, pp. 1–6.

[11] Q.-Z. Zhou, X. Xie, J.-C. Nan, Y.-L. Xie, and S.-Y. Jiang, "Fault tolerant reconfigurable system with dual-module redundancy and dynamic reconfiguration," *Journal of Electronic Science and Technology*, vol. 9, no. 2, pp. 167–173, 2011.

[12] T. Koal, M. Ulbricht, P. Engelke, and H. T. Vierhaus, "On the feasibility of combining on-line-test and self repair for logic circuits," in *2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, 2013, pp. 187–192.

[13] M. R. Boesen, J. Madsen, and P. Pop, "Application-aware optimization of redundant resources for the reconfigurable self-healing edna hardware architecture," in *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 2011, pp. 66–73.

[14] E. Benkhelifa, A. Pipe, and A. Tiwari, "Evolvable embryonics: 2-in-1 approach to self-healing systems," *Procedia CIRP*, vol. 11, pp. 394–399, 2013.

[15] A. Sobe, W. Elmenreich, T. Szkaliczki, and L. Böszörmenyi, "Seahorse: Generalizing an artificial hormone system algorithm to a middleware for search and delivery of information units," *Computer Networks*, vol. 80, pp. 124–142, 2015.

[16] F. J. Doyle, L. M. Huyett, J. B. Lee, H. C. Zisser, and E. Dassau, "Closed-loop artificial pancreas systems: engineering the algorithms," *Diabetes care*, vol. 37, no. 5, pp. 1191–1197, 2014.

[17] P. C. Haddow and A. M. Tyrrell, "Evolvable hardware challenges: Past, present and the path to a promising future," in *Inspired by Nature*. Springer, 2018, pp. 3–37.

[18] O. Eldash, K. Khalil, and M. Bayoumi, "On on-chip intelligence paradigms," in *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2017, pp. 1–6.

[19] O. Shehory, "Shadows: Self-healing complex software systems," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2008, pp. I–71.

[20] D. Breitgand, M. Goldstein, E. Henis, O. Shehory, and Y. Weinsberg, "Panacea towards a self-healing development framework," in *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, 2007, pp. 169–178.

[21] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems*. IEEE, 2008, pp. 1192–1195.

[22] C. M. Fonseca, P. J. Fleming *et al.*, "Genetic algorithms for multiobjective optimization: Formulationdiscussion and generalization." in *Icga*, vol. 93, no. July. Citeseer, 1993, pp. 416–423.

[23] P. K. Rajput and G. Sikka, "Exploration in adaptiveness to achieve automated fault recovery in self-healing software systems: A review," *Intelligent Decision Technologies*, vol. 13, no. 3, pp. 329–341, 2019.

[24] S. Ghahremani and H. Giese, "Evaluation of self-healing systems: An analysis of the state-of-the-art and required improvements," *Computers*, vol. 9, no. 1, p. 16, 2020.

[25] S. Ghahremani, H. Giese, and T. Vogel, "Improving scalability and reward of utility-driven self-healing for large dynamic architectures," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 14, no. 3, pp. 1–41, 2020.