Trust is good, Control is better: Hardware-based Instruction-Replacement for Reliable Processor-IPs

Kenneth Schmitz*† Arun Chandrasekharan* Jonas Gomes Filho* Daniel Große*† Rolf Drechsler*†

*Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
†Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{kenneth, arun, gomes.filho, grosse, drechsler}@cs.uni-bremen.de

***Abstract*—Fault-free function and defect tolerance are key requirements for modern embedded systems. To meet time-to-market constraints, complex IP-components are used to assemble even more complex semiconductor products. Often, trust is required since these IPs are developed, verified and tested by external third-party IP-providers. In this work, we focus specifically on processor-IPs. A method for run-time instruction-replacement on hardware-level is presented to increase the reliability of the system. In contrast to existing techniques, our scheme can easily deal with black-box components and is comparatively lightweight. Furthermore, it includes an easy to use methodology for automated and convenient implementation. The results shows the successful application of this novel technique for reliable integration of state-of-the-art RISC-based processor-IPs.**

I. Introduction

Modern state-of-the-art semiconductor systems experience a rapid growth in terms of complexity. Recent studies have shown that the industry moves towards larger designs and at least 17% of all surveyed designs incorporate 500 million transistors or more. As a matter of fact, a large portion of the development time is spent on verification, i.e. almost 60% are confirmed in a recent world-wide study summarized in [8]. Since time-to-market and verification represent conflicting objectives in this context, it is obvious that the development of the next-generation systems requires reuse of existing functionality. This is typically realized by in-house or third-party *Intellectual Property* (IP)-components. At this point, the IP-integrator has to *trust* the IP-provider's verification strength and quality. If the IP-provider did a good job in verifying the IP, it can be used reliably and the IP-integrator can spend his time and resources on parts of the system that will provide differentiation.

Since the IP is essentially a black-box from the integrator's perspective, dealing with IP related errors increases the effort of integration. This is the case for processor-IPs in particular. In general, depending on the category of errors (e.g. design bug, soft error, aging), different counter-measures exist for mitigation. With this work we focus on the first class, i.e. functional design bugs on the ISA-level which may have been missed during the verification phase of the IP-provider. A well known example is the "FDIV" bug from the Pentium processor. Addressing these design-bugs on a software-level remains a hard problem. It often requires hardware-support (e.g. [21]) otherwise it will yield a significant observation overhead as shown in [6]. Furthermore legacy-code often prevents simple software-based solutions as some code-portions are beyond the software-developer's control or knowledge, hence this code should not be modified afterwards. Hence our scheme is fully transparent to the executed software.

If the problem has not been solved at design time, the only remaining approaches are those which ensure correctness at run-time. In this field most of the earlier published work require detailed insight to the design-data (for more details we refer to the related work section) which contradicts to the third-party principle of IP, where no sources are available. As a consequence, alternatives have been investigated. Primarily, they propose the encapsulation of the IP to "control" (or fix) the nested component or the surrounding system. In [3] a "shield" is synthesized which continuously monitors the input/output of the design and corrects its erroneous outputs. This approach, however, is impractical for complex processor-IPs since a complete property set for the processor is required. A more general approach has been proposed in [7]. The paper introduced the concept of a container to be installed around the IP-block. The container then protects both, the IP and the environment. As a concrete application the concept was demonstrated for monitoring and fixing of bus protocol glitches by essentially synthesizing the respective logic from a property specification. Based on this concept the work of [5] showed how to counter a memory disturbance attack following an extended synthesis strategy.

In this paper we propose a methodology which still follows the container principle, but focuses on more complex processor-IPs. For them, the synthesis-based approaches cannot be used, since a complete property specification of a processor core is unrealistic and the synthesis results would become far too complex. In essence, our idea for reliable processor-IPs integration is to replace failing instructions at run-time by another set of instructions which gives the equivalent functionality. We implement this generic scheme for a *Reduced Instruction Set Computer* (RISC)-based processor-IP. It matches the next executed instruction and replaces it instantaneously by a pre-defined replacement-code.

Our solution is generic in the sense that the replacement is fully software-specified and it can be provided by programmers, i.e. the fix is specified in form of instructions and hence no detailed knowledge of the processor core is required. Moreover, for user convenience, we provide a *Domain Specific Language* (DSL), which is directly used to generate the robust and reliable replacement mechanism. Overall, our methodology enables reliable processor-IPs by a hardware-based replacement approach for fixing erroneous instructions.

This work was supported in part by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1 and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative. This work was also supported by the DAAD.

To summarize, the main contributions of this paper are:

- Convenient IP-integration methodology for fixing erroneous instructions of processor-IPs
- Specification of correction scheme on the programmers level using a DSL
- Lightweight and very cost-effective approach due to its low implementation overhead

The remainder of this paper is structured as follows. In Section II related work is discussed. Our methodology will be introduced in Section III. Subsequently, in Section IV our method is demonstrated for a modern RISC implementation. The achieved results are discussed in Section V. Finally, Section VI concludes our paper.

II. Related Work

There have been several approaches proposed for design and use of reliable processor-IPs. Many of these approaches require design modification and/or detailed internal knowledge of the processor-implementation. An overview of the relevant works is given next.

In [14] the multi-compartment concept introduces multiple protection domains, which enable secure sharing of processing, memory and other system resources. The approach was extended consequently for more complex multi-core and shared memory architectures [13].

A programmable error-detector was integrated to a processor design in [12] and [15]. Errata signatures and internal signals are used to detect erroneous instructions and insert appropriate measurements. The authors propose built-in mechanisms to avoid an invalid or incorrect system-state.

A state-matcher is implemented in [18] as a part of a *Field Repairable Control Logic*, to derive the internal states of the processor. This is used to correct design errors and circumvent wrong computations by switching the processor to a degraded execution mode. Additionally, the authors provide a detailed discussion on a minimum set of signals needed to determine the processor's internal state.

In [17] the framework "CASPAR" has been presented. The framework deploys hardware-detectors inside the cache-subsystem to assess the system state. This approach specifically deals with problems related to multi-core processors as they often partially share their caches. The idea was shown to be effective and very lightweight with respect to logic overhead and performance reduction.

There have also been hybrid approaches that combine software and hardware such as [16] for a VLIW processor with a statically scheduled data path.

As mentioned before, most of the approaches proposed so far require some added functionality to the processor core or monitoring of internal signals of the processor. However, in practice, this is mostly not viable due to the cost, effort and licensing restrictions. In contrast to these white box approaches, our work treats the processor-IP as a black-box and no internal signal information or extra functionality is assumed. Moreover, our approach is automated and the modifications introduced are transparent to other components of the system and software, thereby vastly extending the applicability.

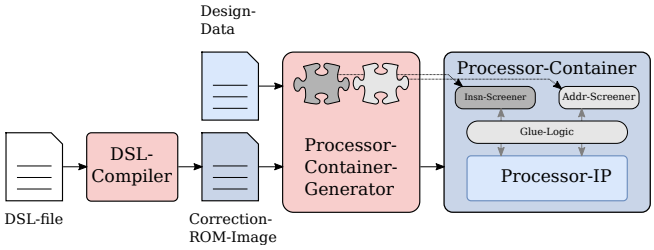

Fig. 1. Processor container generation flow

III. Methodology

Our approach for integration and reliable operation of the processor-IP consists on replacing an erroneous instruction at run-time by another functionally equivalent set of instructions to "work around" the design bug and related effects. This is achieved using a hardware encapsulation-scheme called *processor-container*, which is generated from the requirements specified in a DSL. In the following we give a brief overview on the general flow for the generation of the processor-container. After this, we explain each component in detail.

The overall flow is depicted in Figure 1. Based on the *Instruction Set Architecture* (ISA) of the processor-IP, the user defines the erroneous instruction and the correction in the DSL. Subsequently, the DSL-compiler generates a *correction-ROM-image*, which holds the replacement-code. The flow further proceeds to the processor-container generation. Here, the container-generator processes the design data (e.g. the interfaces of the processor-IP) and the correction-ROM-image from the DSL. Both are assembled and the additional circuitry (building-blocks for detecting and correcting; depicted in form of puzzle-pieces in Figure 1) is added to generate the final processor-container.

The whole approach is transparent to the software and to any other IP involved in the system. Hence, this encapsulated processor-IP serves as a "drop-in substitution" in the conventional system design flow. This is an important aspect of our work that clearly differentiates it from previous solutions reported in the literature.

A. DSL

The primary purpose of the DSL is to provide an easy-to-use replacement mechanism. A DSL should have two essential components; a *detection* segment and *correction* segment. The segment for detection essentially answers the question - *"what is the behavior to be detected?"*, and the correct section is the solution to this problem; *i.e. "how to correct this error if its detected?"*. It does not provide the full flexibility of a programming language, but rather it enables the IP-integrator to specify the requirements concisely.

In this work, the DSL is created towards detecting erroneous instructions and their correction. Hence, we provide the convenient abstraction from hardware and machine-encoded binary in terms of the DSL and assembly. This approach has several advantages, since the IP-integrator does not need to know the complex details of the processor and the specific instruction-encoding/decoding schemes. Moreover, this is impossible in the first place, since the processor-IP is assumed to be a black-box. Besides, the implementation details are different in each generation of processor-IP, but the ISA remains consistent. Also the correction and detection approaches can be easily described

from a procedural way similar to a software application. Moreover, describing hardware directly is a very challenging task with considerations on concurrency and specific hardware architecture. The DSL-compiler addresses all these concerns. In short, for our methodology the knowledge of the ISA and basic assembly syntax are the only prerequisites from the IP-integrator's standpoint.

The DSL-compiler processes the input description and generates the correction-ROM-image. The circuitry to match the erroneous instruction, is generated later by processor-container-generator. The typical pattern of a DSL is shown below.

```
DETECT: /* erroneous instruction */
CORRECT_BEGIN:
        /* replacement code segment */
        /* Procedure:
           check assembly code
           backup assembly code
           alternate compute assembly code */
CORRECT_END:
```

Listing 1. DSL skeleton for container-generation

At first glance, it may look that using assembly instead of a complete software language (e.g. C/C++) restricts the scope of the container. However, it has to be noted that the DSL-compiler directly generates the correction-ROM from the assembly. Hence, this DSL-based approach has several advantages in direct comparison to describing the matching-hardware in an HDL. Moreover, since assembly grants full control over the processor (e.g. registers, control flow, etc.), a combination of assembly, embedded to a concise DSL-syntax, is the best fit.

B. Container-Architecture

The container-generator processes the information from the DSL-compiler together with the original design-data (e.g. interfaces) to generate the final encapsulated IP-core. This container incorporates all interface requirements of the processor-IP and implements the necessary hardware for detection and correction of an erroneous instruction.

To implement this functionality in a modern state-of-the-art processor[1] is a challenging task. It is also important to note that the processor-IP is treated as a black-box and no information about the internal states is externally available. The main complexity arises from the fact that any instruction sequence observed at the memory interface is not necessarily executed. During normal operation, these instruction streams are cached by the included cache hierarchy of the processor. Data is rather loaded from this local memory (integrated as part of the processor hardware, hence, not visible from the outside) than from the main memory. Furthermore, if there is a cache-miss, the processor transfers a complete memory-page (as opposed to a single instruction) in a series of memory burst-read cycles to the cache. Hence, observing an erroneous instruction at the memory interface does not imply that it is eventually executed. Additionally, there is a high amount of non-determinism at this level. There are events like interrupts and exceptions that alter the normal flow of execution in the processor significantly.

To cope up with this uncertainty, we have introduced a two tier architecture in the form of *screeners* as part of the container. The first one, namely *instruction-screener*, scans for the erroneous instruction. The second one is the *address-screener*. It scans if the address range lies within a dedicated pre-defined memory range which holds the correction-ROM. At the current state of development, highly specialized or non-standard *interrupt service routines* (ISR) are not entirely covered by our approach which will be addressed in future work.

[1]A modern state-of-the-art processor at least contains integrated cache and pipeline stages.

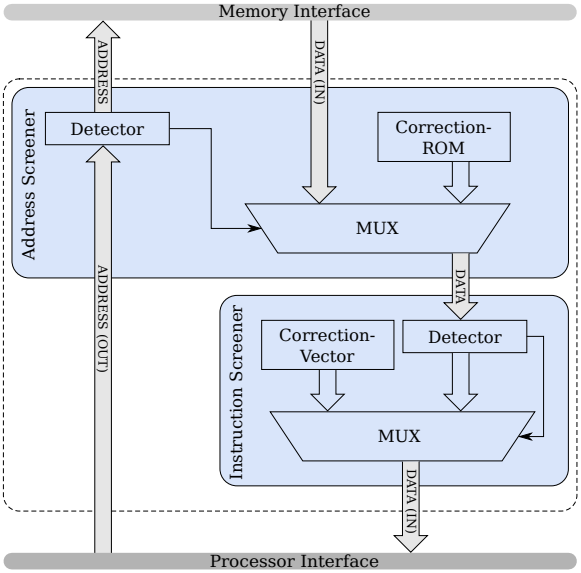

Fig. 2. Instruction- and address-screeners

The instruction-screener monitors a sequence of instructions, but it can only do a one-to-one replacement of the instructions due to the non-determinism explained before. Hence, we explicitly disallow any internal synchronization between the screener instances inside the container. The instruction-screener and the address-screener act independently, but the combined action creates the replacement-mechanism. In this way, the processor is restricted (or allowed) only to work with the information provided by the container controlled environment.

1) Instruction-Screener: The instruction-screener scans the incoming instruction stream and replaces an erroneous instruction with a vector-jump to the location of the correction-ROM. This is a *same-cycle*, *instruction-to-instruction* replacement scheme and essentially achieved by using hardware comparators and multiplexers only. Since interruptions (e.g. interrupts, exceptions, etc.) could modify the sequence of execution at any time, obviously this vector-jump must not span more cycles than the erroneous instruction at most, as such a scheme would affect the spatial and temporal consistency of the instruction stream. In Figure 2, the instruction-screener is shown. It is a lightweight hardware logic that monitors the data-path to the processor.

2) Address-Screener: This component scans the address bus for the applied address. Since our proposed scheme suggests the code execution from a correction-ROM, the execution has to be redirected to this ROM. In typical scenarios, the location of the correction-ROM is moved to an un-populated area inside the addressable address-space. If the vector-jump causes the processor to jump to the address of the correction-ROM, the address-screener instantaneously feeds the content of the correction-ROM to the processor. This ROM is compiled to have the correction routines as given in the DSL. Similar to

instruction-screener, the address-screener is also a multiplexer logic that switches to a different set of instructions (from the correction-ROM).

3) Container-Action: The container will engage into operation only when the replacement is needed. In all other situations, it will act as a passive monitor. The sequence of operations leading to the correction of an erroneous instruction can be explained as follows. If the executed instruction-stream contains an erroneous instruction, the instruction-screener swaps it with a vector-jump. If the processor eventually executes this instruction, the control is transferred to the correction-ROM address-range. Simultaneously, the address-screener monitors the address, and as long it detects this address-range, the data-path is re-routed to take instructions from the correction-ROM. The correction-ROM is always compiled by the DSL-compiler in the in form of a standalone (self-contained) function. Additional control transfer mechanisms are augmented to this correction-ROM as part of the compilation in order to redirect the control back to the original instruction-stream once the correction is completed. Thus, using instruction- and address-screener, the erroneous instruction is omitted from execution since it will be consequently removed from the instruction-stream to the processor-IP. In other words, the container modifies the *environment* in which the processor operates, rather than the processor logic itself.

Although the proposed methodology is fairly easy to use, some important pitfalls, when attempting a replacement scheme like this, need to be noted. First of all, the instruction-screener can monitor a sequence of instructions, but can only do a one-to-one replacement of the instructions. Therefore, if additional house keeping is required (e.g. saving the state of some registers, or checking the status of some operation), it has to be incorporated as part of the correct section in the DSL. However, this is not a limitation by itself, since the designer can be provided with a pre-built library of functions. These functions can contain code for setting up the safe environment for the replacement-code or restoring the initial state after the replacement. Moreover, the retriggering of erroneous instructions in the correction phase has to be dealt with. Since there are a wide range of external interrupt- and exception-mechanisms, the re-triggering of erroneous instructions as part of these has to be always considered as a possibility in a real situation.

For successful application of our methodology, two criterion for the target processor core have to be met. First it should support an unconditional control transfer instruction to a reserved memory area where the correction routines are stored. Second it should have a convenient mechanism to save the address of the next instruction (the one immediately after the jump). This address will be later used to return from the correction routine. These two conditions hold for many of the modern RISC processor architectures such as MIPS, SPARC, ARM, RISC-V, PowerPC etc.

At this point, it is important to add a note on generality of our approach. This methodology is applicable with little or minor modifications in a variety of scenarios. Building *Triple Modular Redundancy* (TMR) schemes for reliable hardware using replication in time [9], is fairly straight forward with this approach. In a similar way, the *detect* section given in the DSL does not necessarily need to detect an *erroneous* condition. However this is not the main focus of this paper and a detailed discussion on such capabilities is omitted for the sake of brevity.

IV. Case Study

We use the RISC-V processor architecture [19] for the demonstration of our methodology. General features of the RISC-V architecture are 32 general purpose registers, 64-bit address space (supports other formats also) and architectural support for position-independent code. Using this architecture is motivated by several reasons. RISC-V is a modern, high quality, general purpose ISA based on RISC principles. The ISA is designed to handle a wide variety of modern high end embedded systems and computer devices [10]. The architecture is open source and hence it is particularly suited for academic work. Further open source implementations of this ISA are available [11], [4] with a complete set of tools including software compilers.

We base our case study on one such implementation IP called *Rocket-chip* [2]. Rocket-chip is a 64-bit, 5-stage single-issue in-order pipeline processor. The design has separate L1 and L2 memory caches with 64 entry *branch target buffer* (BTB), 256 entry *branch history table* (BHT) and 2 entry *return address stack* (RAS). The memory management unit supports page-based virtual memory addressing, support for DMA and other high speed interfaces. Furthermore, this implementation comes with a facility to automatically generate a cycle accurate emulator[2] along side with the hardware and is described in *Chisel* language (Constructing Hardware in SCALA Embedded Language [1]). Chisel supports advanced hardware design using parametrized generators, functional programming and object orientation, and this aids greatly in the development flow.

For demonstration of this case study, we reserve two registers from the general purpose register list of Rocket-chip. The first register serves as the *link* register, the register where the return address after a successful correction is stored. By design, the processor updates this register when a branch is taken [3]. The second register holds the base address of the correction routine. This is required by the architecture of RISC-V since in this architecture the branch instruction is always relative to the contents of a register. A direct consequence of this is that the software compiler, targeted for this methodology, has to generate binary which leaves two registers of the complete register list unoccupied. However, this limitation arises only for the sake of demonstration.

Our experimental setup, as shown in Figure 3, is organized as follows. An arbitrary instruction in the Rocket-chip is assumed to be erroneous and this is specified in the DSL along with the correction code. This is given as the input to our DSL-compiler that generates the container description. The container-generator generates the encapsulated processor-IP based on this description. As part of this experimental setup, a cycle accurate emulator is also generated along with the processor-container IP core. We use this emulator for all the experimental purposes. In our setup all the process up to the emulator generation is fully automated and the user has to provide only the DSL. A cross compiler[4] is also built based on this description which can then be usedto compile software programs targeted to

[2]This emulator is generated as part of the build flow.

[3]In RISC-V, any register can be provided as the link register.

[4]We use GNU Compiler Collection - v5.3 in this work.

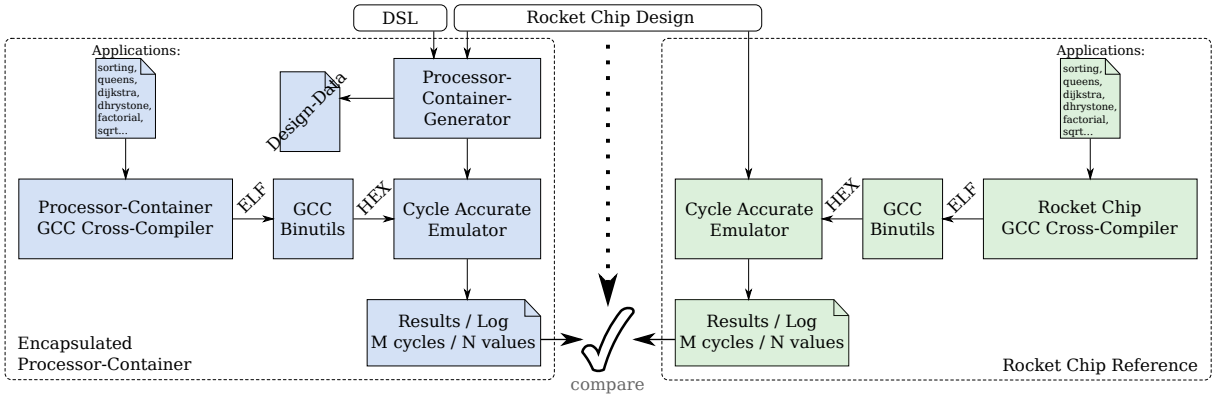

Fig. 3. Complete Experimental Setup

this encapsulated processor. In a similar fashion we build the cycle accurate emulator and the cross compiler for the original Rocket-chip processor.

This Rocket-chip processor, the associated emulator and the cross-compiler serve as the golden reference tools throughout our experiments. The only difference between the processor-container cross-compiler and the Rocket-chip cross-compiler is that the former one uses only 30 registers instead of the full 32 registers for application binary generation.
As mentioned before, in this case study two registers are reserved exclusively for the container-action and hence this reduction in the number of registers.

A wide variety of software applications were chosen from standard benchmark suits and other common programming tasks and two flavors of binaries are generated out of these; one with the Rocket-chip cross compiler and the other with the encapsulated processor-container compiler. These application binaries are in the ELF format[5] and these are converted to HEX format (ASCII text form of the binary) using GNU Binary Utilities[6]. The cycle accurate emulators reads in this HEX file and executes the application. Thus as shown in Figure 3, a virtual emulator platform is provided where compiled applications can be run and compared against each other.

V. Discussion of Results

The robust integration of the processor-IP was discussed such that the main interest for the case study is, to show that the processor-container scheme works transparently in a variety of work loads. Hence we chose the integer *multiply* instruction from the RISC-V ISA as the erroneous instruction. The multiply or *mulw* is a fairly repeated instruction in many common programs and benchmarks and therefore a huge number of hits for the container action is anticipated for bug-compensation. A looped addition-routine is provided inside the DSL as the correction mechanism for the erroneous multiply instruction.

As expected, the implementation of this addition serves solely the demonstration purpose. Based on the input-parameters, the anticipated run time will easily increase. In case of time-critical applications (e.g. real-time- and safety-critical-systems), more efficient replacements (e.g. common Russian Peasants multiplication) can be implemented.

Since this replacement is parameter-independent wrt. the time-domain, the linear overhead can be considered in advance. This will additionally serve as a stress test with the container being active in several places with different program conditions. Hence, the successful completion of programs with the *mulw* being substituted with looped additions by the container, is demonstration such that the container-logic is effective. The resulting emulator execution trace for the reference Rocket-chip is compared with the processor-container emulator trace to check for the successful and corrected completion of the execution.

The execution results for some of the common programs is summarized in Table I. All programs are executed until completion and results are verified to be equivalent. The number of *mulw* instructions (n) in the application is provided along side with the program name. The main comparison in this table

TABLE I
Results Summary

Application Details		Reference	Processor-container			
		Rocket	Total†		Correction‡	
Program	mulw n	cycles c	cycles $\hat{c}$	factor	cycles $\tilde{c}$	%
multiply[1]	20	1594	2888	1.812	936	32.41
sqrt1[2]	20	6057	7847	1.296	1654	21.08
sqrt2[3]	24	6057	8628	1.424	2745	31.82
sqrt3[4]	567	6628	766480	115.642	759712	99.12
factorial1[5]	60	2056	4362	2.122	2258	51.77
factorial2[6]	147	2523	28273	11.206	25772	91.15
scalar[7]	120	4070	8532	2.096	4410	51.69
Dhrystone[8]	2005	437212	507433	1.161	68880	13.57
64-queens[9]	0	5928057	5928507	1.000	0	0
quick-sort[10]	0	884930	886570	1.002	0	0
reverse-sort[11]	0	1409719	1410445	1.001	0	0
towers[12]	0	46267	46799	1.011	0	0
vector-add[13]	0	37243	37763	1.014	0	0
Dijkstra[14]	0	7947	9463	1.191	0	0

† Total number of cycles ($\hat{c}$) inside processor-container.
‡ Relative number of cycles ($\tilde{c}$) inside correction-ROM.

1: Plain multiplication of two numbers, no further processing.
2, 3, 4: Different input for the Babylonian square-root algorithm.
5, 6: Different parameter values for the factorial program.
7: Computation of dot- and scalar-product.
8, 9: Dhrystone Benchmark [20], 64-queens problem
10, 11, 12: Towers of Hanoi, quick sort and reverse sort programs
12, 13, 14: Vector addition and shortest path algorithm.

[5] *Executable and Linking Format* (ELF) is the default format for executables and libraries in a Linux Operating System.

[6] GNU Binutils are the standard tools used in conjunction with GNU Compiler Collection.

is the number of cycles taken by the reference Rocket design (c) and the cycles taken by the processor-container (including CPU, $\hat{c}$). Besides the number of cycles inside the container ($\hat{c}$), those executed as part of the correction procedure (in the table $\tilde{c}$) are also provided in the column "Correction". The increased number of cycles as a result of container-presence and the percentage of cycles executed as part of correction are also shown in the table. One important aspect to consider is that even though the software compilations are configured differently for the Rocket-chip and processor-container variant (see Figure 3), the number of mulw instructions in both final binaries is found to be the same. Hence we do not provide these separately.

The first set of applications do extensive stress-test to the processor-container as they heavily rely on mulw instruction. It is notable, that the number of cycles taken by the processor-container has increased in most of the cases. This has to be expected when a single *mulw* instruction (single cycle) is substituted with a series of instructions and associated load/store mechanisms. Next, we examined the standard Dhrystone benchmark for embedded systems. The percentage of instructions executed as part of correction routine is about 14%. In addition we have selected a set of applications which do not contain *mulw* instructions, to study the interference incurred due to the container mechanism. Since these are similar but effectively different emulators and differently configured compilers, some difference are anticipated. However, in this case study the software compiler of the processor-container cannot use the full register-set compared to default Rocket-chip configuration. This will have some performance impact, which is reflected in Dijkstra algorithm for example. Either way, we demonstrate the successful embedding of a black-boxed processor-IP (i.e. RISC-V). The minor modification we introduced for the implementation of this work, can be subsequently ruled out as soon as this methodology is established.

Finally, we synthesized the processor-container in Xilinx FPGA and the results are given in Table II. From the table it is evident that the overhead due to the container is minimal and this confirms the lightweight nature of our approach.

VI. Conclusion

This paper proposed a methodology for reliable black-box processor-IP integration based on the principle of run-time instruction replacement. The user specifies the erroneous instruction and the correction in an easy-to-use DSL at design-level. Results show the effectiveness of the proposed solution for a state-of-the-art RISC processor. Our approach is lightweight and only introduces minimal hardware overhead. This way, we can overcome the notion of trust towards the IP-developer's verification and test methodologies and implement robust systems composed from IP components.

TABLE II
Hardware Resource Utilization

Element	Rocket chip	Processor-container	Increase
Registers	14086	14096	0.07%
LUTs	36906	37173	0.72%
Logic	31461	31728	0.85%

Results from Xilinx FPGA Virtex6 (XC6VLX75T) using ISE 14.7

References

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conf.*, pages 1212–1221, 2012.

[2] S. Beamer, H. Cook, Y. Lee, S. Twigg, H. Vo, and A. Waterman. rocket-chip, 2016.

[3] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang. Shield synthesis: - runtime enforcement for reactive systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 533–548, 2015.

[4] C. Celio, D. A. Patterson, and K. Asanović. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical Report UCB/EECS-2015-167, EECS Dept., University of California, Berkeley, Jun 2015.

[5] A. Chandrasekharan, K. Schmitz, U. Kühne, and R. Drechsler. Ensuring safety and reliability of IP-based system design - A container approach. In *IEEE International Workshop on Rapid System Prototyping*, pages 76–82, 2015.

[6] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *International Symposium on Computer Architecture*, pages 377–388, June 2008.

[7] R. Drechsler and U. Kühne. Safe ip integration using container modules. In *International Symposium on Electronic System Design*, pages 1–4, 2014.

[8] H. D. Foster. Trends in functional verification: a 2014 industry study. In *Design Automation Conf.*, pages 48:1–48:6, 2015.

[9] T. Koal, M. Ulbricht, and H. T. Vierhaus. Virtual TMR schemes combining fault tolerance and self repair. In *Euromicro Conference on Digital System Design (DSD)*, pages 235–242, 2013.

[10] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanovic. A 45nm 1.3 ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *European Solid State Circuits Conference*, pages 199–202. IEEE, 2014.

[11] R. Mullins, G. Ferris, and A. Bradbury. lowRISC: A fully open-sourced, Linux-capable, RISC-V-based SoC, 2016.

[12] S. Narayanasamy, B. Carneal, and B. Calder. Patching processor design errors. In *Int'l Conf. on Comp. Design*, pages 491–498, 2006.

[13] J. Porquet, A. Greiner, and C. Schwarz. NoC-MPU: a secure architecture for flexible co-hosting on shared memory MPSoCs. In *Design, Automation and Test in Europe*, pages 1–4, 2011.

[14] J. Porquet, C. Schwarz, and A. Greiner. Multi-compartment: a new architecture for secure co-hosting on SoC. In *International Symposium on System-on-Chip*, pages 124–127, 2009.

[15] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas. Patching processor design errors with programmable hardware. *Microelectroics Journal*, 27(1):12–25, 2007.

[16] M. Schölzel and S. Müller. Combining hardware- and software-based self-repair methods for statically scheduled data paths. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 90–98, 2010.

[17] I. Wagner and V. Bertacco. Caspar: Hardware patching for multicore processors. In *Design, Automation and Test in Europe*, pages 658–663, 2009.

[18] I. Wagner, V. Bertacco, and T. Austin. Using field-repairable control logic to correct design errors in microprocessors. *IEEE Transactions on Computer Aided Design of Circuits and Systems*, 27(2):380–393, 2008.

[19] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The RISC-V instruction set manual, volume i: Base user-level ISA. *EECS Dept., UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[20] A. R. Weiss. Dhrystone benchmark. *History, Analysis, "Scores" and Recommendations, ECL/LLC*, 2002.

[21] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture*, pages 122–133, June 2003.