Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 180242, 11 pages
doi:10.1155/2010/180242

Research Article

Proof-Carrying Hardware: Concept and Prototype Tool Flow for Online Verification

Stephanie Drzevitzky, Uwe Kastens, and Marco Platzner

Department of Computer Science, Faculty for Electrical Engineering, Computer Science and Mathematics, Computer Engineering Group, Warburger Str. 100, 33098 Paderborn, Germany

Correspondence should be addressed to Stephanie Drzevitzky, stephanie.drzevitzky@upb.de

Received 8 March 2010; Revised 18 October 2010; Accepted 17 December 2010

Academic Editor: Lionel Torres

Dynamically reconfigurable hardware combines hardware performance with software-like flexibility and finds increasing use in networked systems. The capability to load hardware modules at runtime provides these systems with an unparalleled degree of adaptivity but at the same time poses new challenges for security and safety. In this paper, we elaborate on the presentation of proof carrying hardware (PCH) as a novel approach to reconfigurable system security. PCH takes a key concept from software security, known as proof-carrying code, into the reconfigurable hardware domain. We outline the PCH concept and discuss runtime combinational equivalence checking as a first online verification problem applying the concept. We present a prototype tool flow and experimental results demonstrating the feasibility and potential of the PCH approach.

1. Introduction

Dynamically reconfigurable hardware combines hardware performance with software-like flexibility and finds increasing use in networked systems. The dynamic reconfiguration capability provides networked systems with the flexibility to download new hardware functionality or hardware updates as needed. Since system downtime is often unacceptable, newly received hardware modules would have to be installed and run without previous extensive system testing and verification. However, due to its wide applicability reconfigurable hardware is often used in security and safety critical systems where an unintended system behavior could have severe consequences, such as heavy financial damage, loss of human life, and threat to national security, assuring the absence of unwanted and malicious behavior caused by outside attacks as well as internal construction flaws becomes essential in such scenarios.

The novel contribution of this paper is the presentation and elaboration of *proof-carrying hardware (PCH)* as an approach to reconfigurable system security, substantiated with an in-depth depiction of a prototype tool flow including experimental results. PCH takes a key concept from software security, known as proof-carrying code [1], into the reconfigurable hardware domain. We envision a scenario where an embedded reconfigurable target system, that is, the consumer, requests new functions with certain security guarantees at runtime. In response, reconfigurable hardware modules are being created by a design center, that is, the producer. The producer has to invest the substantial computational resources required to create a proof. The proof is then combined with the reconfigurable module into a binary and delivered as *proof-carrying bitstream*. The consumer can quickly verify the security property and, if successful, instantiate and run the hardware module. In essence, the consumer is enabled to only run verified hardware modules without having to trust the producer or rely on a secured transmission process or having to compute a formal proof of security features.

The PCH concept is rather general and can be applied to many types of security properties and corresponding proof systems and proofs. As a first application of PCH, we focus on runtime combinational equivalence checking (CEC) in this paper. Runtime CEC enables a consumer to verify that a requested reconfigurable module actually adheres to its functional specification and, thus, eliminates a major security risk according to [2].

This paper extends our previous conference publication [3] by presenting an extended discussion of the PCH concept and a more elaborated CEC tool flow, using more test functions to demonstrate the feasibility of the PCH concept, and discussing more detailed and conclusive measurements including, for example, memory requirements.

The paper is organized as follows. In Section 2, we review related work. Section 3 presents the novel concept of proof-carrying hardware and elaborates on trust and threat models. Section 4 focuses on runtime CEC as a first PCH application. The implementation of the CEC tool flow, including experimental results and test functions, is depicted in Section 5. A conclusion and further work including possible other safety properties suited for PCH are presented in Section 6.

2. Related Work

In this section we review approaches to reconfigurable hardware security, an area that has gained interest only recently. An overview of security risks present in every step of the life cycle of reconfigurable hardware is presented by Kastner and Huffmire [2]. Drimer [4] also provides an exhaustive survey of possible attacks, involved parties, stages of life cycles, and possible defenses. For a detailed elaboration on FPGA design security, see [5].

A first step towards security of dynamically reconfigurable hardware is to establish trust in the bitstream transmission. Typically, FPGA bitstreams are minimally secured by checksums and some FPGA vendors even offer built-in hardware support for bitstream decryption and embedded keys. Chaves et al. [6] propose a more flexible approach based on hashing to secure correct bitstream delivery. The same authors also address another aspect of reconfigurable hardware security: they interpret the incoming bitstream to check whether the physical regions on the FPGA that are to be reconfigured match the intended reconfiguration area. Drimer and Kuhn [7, 8] distinguish between authentication and confidentiality and discuss a security protocol that combines both aspects to prevent system downgrades. Similar to this, Badrignans et al. propose a combination of special architecture and protocol to avoid the usage of old configurations in [9]. These techniques, however, assume that the module producer can be trusted and implemented the correct functionality and do not inspect functional properties of the reconfigurable modules at runtime.

Huffmire et al. [10] focus on multicore reconfigurable systems where several cores access the same memory. The authors define a set of high-level policies covering security scenarios such as Chinese Wall and Secure Hand-Off. The designer uses a formal regular language to describe valid memory accesses for a core and the interactions included in the policies. Starting from this formal specification, a synthesis tool generates a memory reference monitor. In [11], Huffmire et al. propose to secure IP cores on FPGAs with physical isolation primitives called moats and drawbridges. While moats prevent unwanted and unanticipated communication between cores, drawbridges allow for controlled and secure communication channels. Drawbridges are especially useful when also a reference monitor is invoked which enforces a memory policy specified for the intended access scenario. The combination of physical isolation primitives and reference monitors and the threats addressed by these techniques are also discussed in [12–14].

Many methods for functional verification known from the domains of software and (static) hardware verification rely on model checking (see, e.g., [15], for a survey). For example, Singh and Lillieroth [16] propose the Core Verification Flow which conducts a decomposition of a core and extracts a reference design for each entity to be verified from the core's behavioral specification. Eventually, they receive a logical formula against which the implementation is compared with NP-Tools or Prover Software. The Core Verification Flow runs the complete runtime and resource consuming analysis and security verification on the system that also executes the core.

The PCH approach presented in this paper shifts the computational burden for establishing security to the producer of a module which allows us to provide security also to embedded consumer platforms with limited computational power. Moreover, the PCH approach can be used for a multitude of nonfunctional and functional properties of reconfigurable hardware modules, including the above-mentioned ones.

3. The Transition from Proof-Carrying Code to Proof-Carrying Hardware

In this section we first introduce the principles of proof-carrying code and then transfer the concept to the reconfigurable hardware domain. Finally, we elaborate on trust and threat models that can be covered by proof-carrying hardware.

3.1. Principles of Proof-Carrying Code.

3.1. Principles of Proof-Carrying Code. In 1996, Necula and Lee proposed Proof-Carrying Code [1] as a means to validate code from an untrusted source before execution. The scenario includes a code consumer and a code producer. Both have to agree on a safety policy that includes formalisms to capture characteristics of the consumer's execution platform (e.g., machine model, memory accesses, type definitions) and to describe the desired safe code behavior. Depending on the actually chosen formalisms, producer and consumer must also decide on a corresponding proof system.

The producer then creates the program for the required functionality and, depending on the safety policy, may annotate the code with preconditions, invariants, and post-conditions. This step is crucial, since the annotated code together with the safety policy is transformed into the so-called safety predicate. The safety predicate is then proven to hold true in all states of the program. The proof is combined with the code into the proof-carrying code delivered to the consumer. The consumer verifies the received proof and, thereby, checks the code's compliance with the safety policy.

The essence of proof-carrying code is to shift the burden of verification from the consumer to the producer, leaving the consumer with simply checking the delivered proof against the code, a task of insignificant size compared to the actual computation of the proof. Thus, proof-carrying code techniques are of special interest for target systems (consumers) with limited computational resources or systems that need to quickly extend their functionality by downloading mobile code.

The great universality of the proof-carrying code approach has been demonstrated in several case studies. For example, in [1] a subroutine scans incoming network packets and determines whether they should be accepted as being valid or rejected. The safety policy has to secure memory safety and guarantee the termination of the subroutine. Memory safety is ensured in this case by requiring that a designated argument register holds the start address of a package in memory and another register contains the package's size. A further case study computing the checksum of an IP packet extends the concept to loops dealing with memory arrays of varying sizes. Another application of proof-carrying code is demonstrated in [17], where the safety policy not only captures memory access but also handles resource usage, for example, execution times of agents.

The fundamental principle behind proof-carrying code states that for many computational problems validating a given solution is less complex and thus less costly in terms of computations than creating the solution in the first place. From this perspective, the proof-carrying code concept can be extended beyond its original, narrow definition and be applied to properties of interest other than safety features. Furthermore, a solution does not necessarily have to carry a proof in the literal meaning. For example, in [18] Klohs and Kastens apply the proof-carrying code concept to program analysis in compiler contexts: an optimizing compiler performs deep and complex analyses on the code to establish properties which are used as preconditions for optimizing transformations. Such analyses, for example, data flow analyses, are too time-consuming to be performed by a just-in-time compiler at program execution time. Hence, the producer generates intermediate code and annotates it with results of program analysis, thus proving certain properties of the code. At runtime, the virtual machine takes on the role of the code consumer and checks the annotations with respect to the code. If the check passes, the runtime system uses the annotations for optimizing transformations. In [19], Klohs extends the approach to interprocedural program analysis.

3.2. Proof-Carrying Hardware. We propose proof-carrying hardware as the equivalent of proof-carrying code for dynamically reconfigurable hardware systems. Reconfigurable hardware systems often have limited computational resources, rely on a fast instantiation of newly downloaded hardware modules, and are increasingly security and safety critical. All those characteristics match exactly the characteristics of target systems for which proof-carrying code has been established. There are, however, notable differences between software and hardware. Arguably the most important one is the complexity of the machine model. Software executes on instruction-set processors which allows us to abstract code as a sequence of instructions accessing a rather small set of microarchitectural components. In contrast, reconfigurable hardware modules utilize large numbers of spatially arranged (placed and routed) components. Furthermore, while downloaded code plugs into rather matured software ecosystems, there are no standardized or even commonly used execution environments for reconfigurable hardware modules. Lastly, while security and safety concepts for processor-based systems have been studied for quite some time, the field of reconfigurable hardware security is just emerging.

Figure 1 shows an abstract proof carrying hardware scenario. In this scenario, we denote embedded target platforms that execute functions in software and hardware on dynamically reconfigurable heterogeneous architectures (e.g., CPU/FPGA systems) as *consumers*. The functions are created in form of modules by design centers, denoted as *producers*. In this work we focus on the hardware functions that are requested by consumers, created as reconfigurable modules including security proofs by producers, downloaded over some network, and verified, configured, and executed by consumers—all at runtime. Depending on the use case, there are several reasons for the consumer to request a new hardware module, including bug fixes, using optimized functions and, most importantly, user-initiated extensions of the functionality of the embedded target system. The consumer and potential producers must have agreed on and use a common formalism to describe the functional and perhaps also nonfunctional specification of the requested modules, the characteristics of the target architecture, the safety requirements, and the used proof system.

Computing proofs for all tentative reconfigurable modules, security properties, and target architectures in advance is uneconomic. Along the same line we consider it infeasible for the consumer to store statically verified bitstreams for all tentative reconfigurable modules and security properties.

3.3. Trust and Threat Models. For the proof-carrying hardware scenario, we can establish a general trust and threat model similar to Myagmar et al. [20]: as with proof-carrying code, we neither need to trust the producer of the hardware module nor do we require specially secured transmission. The major difference between proof-carrying code and PCH is that the safety policy for the latter targets hardware instead of software modules. PCH considers hardware reconfigurations as entry points for possible security breaches and attacks. The reconfiguration bitstream could contain a module that provides unspecified, additional, or otherwise undesirable functionality. The cause for this might be an intentional attack or an unintentional alteration of the module's behavior during design or transmission.

The consumer sets up a safety policy that may capture not only the functionality of the module but also non-functional properties, for example, the minimum clock frequency and

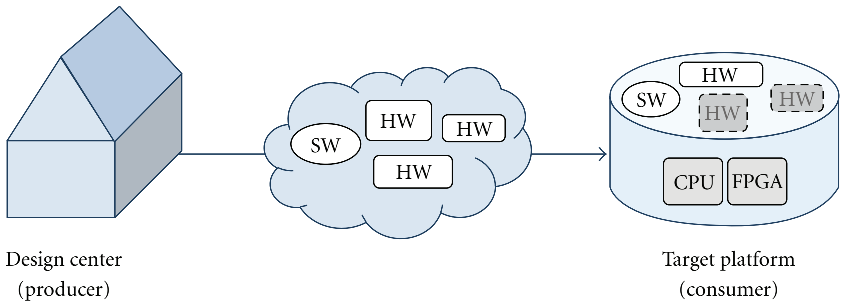

Figure 1: Proof carrying hardware scenario.

area of the module, and the specifics of the reconfigurable platform, for example, the device type and the size and position of the reconfigurable areas. Functional properties can include the complete behavior of the module checked by combinational or sequential equivalence to its specification, or some partial behavior such as ensuring that invariants always hold, accesses to external memory stay within given address bounds, or that bus protocols are adhered to. Any security threat that is described in the safety policy can be dealt with. It is the responsibility of the consumer to ensure that the safety policy is complete in the sense that all security threats to be eliminated for the specific module are covered. Naturally, the possible security threats and thereby results of a malicious hardware module will vary for each host system and use case and might include loss of confidentiality, data theft, data manipulation, or even a complete system failure.

Proof-carrying code and PCH as well do not rely on secure transmission. If the proof check on the consumer side succeeds, it is guaranteed that (i) the proof matches the code and the consumer's safety policy and (ii) the code has the proven property. The check will fail if the proof has been damaged, if it does not fit to the code, or if the code does not establish the required properties. Such a failure may be caused by an accidental or malicious modification of either the code section, or the proof section of the proof-carrying code, or both. In the unlikely case of matching changes to both the proof and the code, there is no damage done since the proof still verifies the code and thus guarantees its compliance with the safety policy. On the other hand, the check will fail if the producer and consumer operate, accidentally or maliciously, with different safety policies. In particular, the proof-carrying code approach is also robust against third parties performing man-in-the-middle attacks, even when circumstances require the safety policy to be public. Since the consumer always knows the original safety policy and uses it to verify the proof, the check would fail if a modified safety policy was used to create the proof. In other words, while it might be possible to pass off a different design, that is, code from a different source, with a matching proof, as soon as the proof check succeeds, no damage is done since any design that complies with the (consumer-stored) safety policy is per definition genuine. The only remaining requirement is a trustworthy procedure for checking the proof.

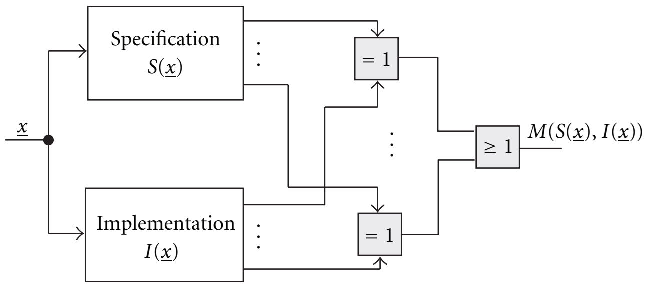

Figure 2: Construction of the miter $M(S(\underline{x}), I(\underline{x}))$.

4. Runtime Combinational Equivalence Checking

Combinational equivalence checking (CEC) is the most fundamental verification problem for hardware. The typical use of CEC is to verify whether a specification of a combinational function, $S(\underline{x})$, is equivalent to an implementation in a specific technology. To that end, the implemented circuit is analyzed and modeled with a logic function, $I(\underline{x})$. Apparently the number of inputs and outputs, respectively, of the specification and the implementation must match. Using $S(\underline{x})$ and $I(\underline{x})$, the miter is formed. The miter is a single-output function that provides both specification and implementation with the same inputs and compares their outputs pairwise with XOR gates. All XOR outputs are then OR-ed together to form the miter, $M(S(\underline{x}), I(\underline{x}))$. Figure 2 shows the construction of the miter graphically.

If under any input $\underline{x}$ the specification and the implementation generate different outputs, the miter will evaluate to 1. Consequently, demonstrating equivalence means to prove the unsatisfiability of the miter. Modern CEC tools represent the miter in conjunctive normal form (cnf) and rely on Boolean satisfiability (SAT) solvers to prove unsatisfiability.

Interestingly, during the last years SAT solvers have progressed into tools that can generate resolution proofs for unsatisfiability. The improvement of such techniques is still a focus of research such as [21–23]. The motivation for this development roots in the desire to build up trust in the results generated by a SAT solver. The direct way would be to prove the correctness of a SAT solver itself, which unfortunately seems to be out of reach given the complexity of modern SAT solvers. The next best approach is to verify

Table 1: Resolution proof trace.

(1)	$x_1 + \overline{x}_2 + \overline{x}_3$	
(2)	$x_1 + x_3$	
(3)	$\overline{x}_1$	
(4)	$x_2 + \overline{x}_3$	
(5)	x_3	Using (2), (3)
(6)	x_2	Using (4), (5)
(7)	$\emptyset$	Using (5), (1), (6), (3)

the unsatisfiability result for each single cnf, a step which requires access to a resolution proof. A resolution proof is a sequence of resolutions on the original cnf and intermediate clauses that eventually leads to an empty clause which models a contradiction. As the size of the generated proof has been a concern, proof traces have been proposed as a compact representation of a proof.

As an example, Table 1 gives a possible proof trace for the cnf $(x_1 + \overline{x}_2 + \overline{x}_3) \cdot (x_1 + x_3) \cdot (\overline{x}_1) \cdot (x_2 + \overline{x}_3)$: the first four lines list the clauses of the cnf. Lines five to seven present the resolution steps and refer to the clauses used to resolve the new terms. In each step, the literal that is used in its negated and in its nonnegated form can be resolved. The resulting clause is the empty clause.

We can make use of resolution proof traces to set up a proof-carrying hardware scenario for runtime (online) CEC. Figure 3 shows the scenario and details the steps producer and consumer perform for runtime CEC. The consumer requests a new reconfigurable hardware module with the combinational function $S(\underline{x})$ and sends the functionality description and the safety policy to a producer. Classically, the producer will run the specification through logic synthesis tools such as FPGA technology mapping and FPGA backend synthesis tools which include place and route and bitstream generation. In the runtime CEC scenario, the producer additionally forms a miter from the specification $S(\underline{x})$ and the implementation $I(\underline{x})$ (synthesized netlist). A CEC tool proves the equivalence and generates the resolution proof trace $P(M(S(\underline{x}), I(\underline{x})))$. Finally, the producer combines the bitstream and the proof trace into the proof-carrying bitstream and sends it to the consumer.

The consumer takes the received bitstream and extracts the implemented logic function $I(\underline{x})$. After forming the miter with this implementation and the original specification, the consumer checks the proof using the proof trace. Only in case the proof holds, the bitstream $B(\underline{x})$ is loaded into a reconfigurable area of the target device.

Any tampering with the bitstream or the proof sections of the proof-carrying bitstream will result in a failed proof check at the consumer side. If both are modified compatibly, the proof check will still fail if $I(\underline{x})$ does not correspond to the original specification $S(\underline{x})$ anymore.

In abstract terms, the safety policy includes the agreement on a specific bitstream format, on cnf to represent combinational functions, and on the use of propositional calculus with its resolution rules to derive and verify the proof. Furthermore, since the resolution proof indexes the clauses of the miter, consumer and producer must use the same way of constructing the miter.

5. Prototype Implementation and Results

To demonstrate the feasibility of runtime CEC (as an application of PCH) we set up the prototype tool flow shown in Figure 4. This proof of concept tool flow is a simplified version of the scenario shown in Figure 3 and serves to validate whether it is possible to shift the verification workload from the consumer to the producer. To this end, it is neither necessary nor the intention of this paper to verify all steps of the production, for example, FPGA backend synthesis tools, to check for correct transmission or to completely implement the consumer's functions. Specifically, the consumer specifies the combinational test function in behavioral Verilog for simplicity. In principle, any other and perhaps simpler specification formalism for capturing combinational functions can be used.

Our prototype tool flow bases on a number of freely available, noncommercial CAD tools which gives us full control over the different parameters involved and file formats used. After receiving the specification, the producer invokes the following tools.

(i) Odin [24, 25] performs front-end synthesis and translates the incoming circuit specification in Verilog into a logic description in blif (Berkeley Logic Interchange Format).

(ii) ABC [26] performs logic optimization and technology mapping to 4-LUTs, generating again a blif file. The optimization is based on And-Inverter Graphs and includes balancing, refactoring and rewriting; all applied multiple times to the circuit with the resync2 command.

(iii) T-VPack [27, 28] packs the LUTs into clustered logic blocks, generating a circuit description in form of a netlist (.net) of such logic blocks. The experiments described in this paper use nonclustered logic blocks and LUTs with four inputs, which are the default values. We run T-VPack to convert the LUT netlist into the format required by VPR and omit further logic specification.

(iv) VPR [27, 28] places and routes the packed logic block netlist and produces placement (.p) and routing (.r) information for inclusion in the final bitstream. The target architecture input file used by VPR has to agree with the specifications used for T-VPack. We use the k4-n1.xml architecture file provided by VPR. We operate VPR in the area minimization mode.

(v) To form the miter cnf, we use again ABC which translates the cnf from And-Inverter Graphs (AIGs).

(vi) The SAT solver PicoSAT [29] at the producer's side proves the unsatisfiability of the miter and generates an extended proof trace file.

(vii) ComPose (own code) is for composing the proof-carrying bitstream.

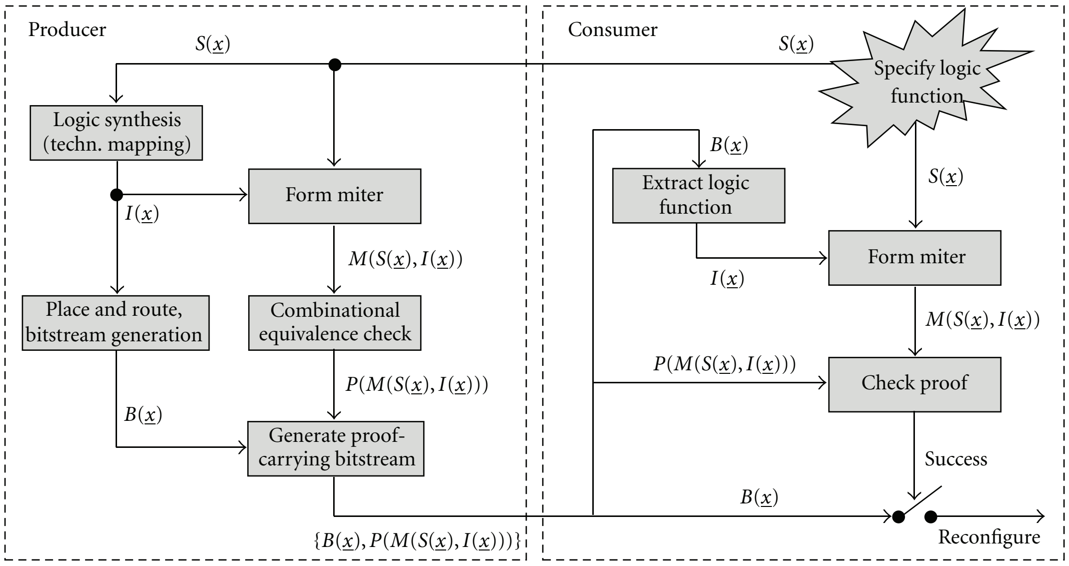
Producer
Consumer
$S(\underline{x})$
Logic synthesis (techn. mapping)
Form miter
$I(\underline{x})$
$M(S(\underline{x}), I(\underline{x}))$
Place and route, bitstream generation
Combinational equivalence check
$P(M(S(\underline{x}), I(\underline{x})))$
$B(\underline{x})$
Generate proof-carrying bitstream
$\{B(\underline{x}), P(M(S(\underline{x}), I(\underline{x})))\}$
Specify logic function
Extract logic function
Check proof
Success
Reconfigure

FIGURE 3: Runtime combinational equivalence checking as a proof of concept study for the proof-carrying hardware concept.

TABLE 2: Runtime measurements for producer and consumer in the online CEC scenario.

Test function	Producer [s]					Consumer [s]			
	Odin	ABC	T-VPack	VPR	PicoSAT	ComPose	DeComPose	ABC′	TraceCheck
Converter	0.187	0.348	0.005	2.040	0.008	0.080	0.004	0.148	0.013
128-bit parity	0.194	0.350	0.100	10.892	0.036	0.111	0.007	0.156	0.027
8-bit add/sub	0.183	0.273	0.007	2.126	0.015	0.034	0.004	0.201	0.010
16-bit add/sub	0.122	0.279	0.011	5.520	0.048	0.074	0.004	0.206	0.035
32-bit add/sub	0.186	0.330	0.012	14.428	0.099	0.154	0.009	0.215	0.059
64-bit add/sub	0.195	0.432	0.220	40.674	0.244	0.327	0.015	0.234	0.126
6-bit multiplier	0.179	0.332	0.008	5.304	0.540	0.145	0.009	0.234	0.483
8-bit multiplier	0.184	0.450	0.010	13.205	15.337	1.148	0.123	0.213	11.179
10-bit multiplier	0.183	0.645	0.014	26.589	256.119	18.849	1.807	0.214	190.630
16-bit multiplier	0.205	1.473	0.040	133.814	3732.031*				
32-bit multiplier	0.229	6.236	0.163	2116.210	3895.797*				
64-bit multiplier	0.527	27.424	0.726	36447.768	6387.947*				

The proof-carrying bitstream is then sent to the consumer that executes the following steps:

(i) DeComPose (own code) for decomposing the proof-carrying bitstream,

(ii) ABC to form the miter,

(iii) TraceCheck [30] to validate the correctness of the proof trace and thereby give the final approval to load and execute the received bitstream; the output is a compact binary resolution trace.

Our simplified prototype tool flow shown in Figure 4 covers the main components and functionalities of the runtime CEC scenario but is incomplete since the consumer does not yet use the original specification to form the miter. Thus, the safety policy in our experiments is the demand for combinational equivalence of the function before and after the logic synthesis/optimization and technology mapping. This is sufficient for the experiments presented in this paper. We use ABC at the consumer side to form the miter from both the unprocessed and the technology mapped and optimized circuit which are embedded in the bitstream. The resulting cnf is then checked for consistency with the cnf in the resolution proof trace. If both cnf forms match and the proof turns out to be correct, the module can be loaded. Completing the tool flow by taking the original specification to form the miter and experimenting with more advanced safety policies is part of future work.

We test the prototype tool flow on an Intel Core 2 Duo 2 GHZ CPU with 4 GB RAM running Linux 2.6.31.5-0.1 and report on results using the following test functions: an EBCDIC to ASCII converter for the letter subset of the EBCDIC code, a 128-bit parity function, n-bit combined adders/subtractors with $n = 8, 16, 32$, and 64, respectively, and n-bit unsigned multipliers with $n = 6, 8, 10, 16, 32$, and 64. We have chosen these test functions and their according input size variations to contrast functions which are presumably rather easy to verify with more demanding ones, that is, the multipliers. The resulting FPGA bitstreams implement all test functions with LUTs; special circuitry such as carry chains or heterogeneous blocks are not yet included in our FPGA model.

Using the prototype tool flow and the test functions we experimentally investigate the following questions.

(1) Does the runtime CEC tool flow work correctly?

(2) What are the runtimes for producer and consumer? The difference between the time required for proving the miter unsatisfiable and generating the proof trace versus the time required to check the proof is of special interest. This time difference constitutes the main savings for the consumer, in comparison to an approach where the consumer runs the complete verification tool chain.

(3) What are the memory usages for producer and consumer? Similar to the runtimes, the savings for the consumer are of special interest.

(4) How large is the overhead for the proof-carrying bitstream? The proof trace increases the size of the bitstream and leads to higher transmission costs.

As a first result, we are able to demonstrate the correct functionality of the runtime CEC prototype. The producer generates FPGA implementations and equivalence proofs for all test functions, which the consumer successfully verifies. Tampering with the files at different stages of the producer tool flow results in the expected effects: modifying the technology-mapped netlist of the miter cnf or the miter cnf itself results in a satisfiable miter, stating that the implemented circuit differs from the specification. Changing the proof trace at the consumer side or during transmission leads to a failed proof check at the consumer.

An important metric for runtime CEC is the required computation time, especially for the consumer. We measure the runtimes of all tools in our prototype. On the producer side, this includes Odin and ABC for creating the blif files as well as the miter cnf, T-VPack and VPR for packing, place and route, PicoSAT for equivalence checking and proof trace generation, and ComPose for compiling the proof-carrying bitstream (consisting of the two blif files, the packed netlist, the placement and routing information, and the resolution proof trace). On the consumer side, we measure the runtime of DeComPose, ABC only building the miter without performing logic optimization, and TraceCheck.

Table 2 presents the runtime measurements for our test functions. Columns two to seven give the computation time for the producer side of the tool flow while columns eight to ten depict the runtimes for the consumer. The table clearly shows that the multiplier test functions form a separate group. First, with growing number of inputs multipliers synthesized from LUTs become huge functions which is demonstrated by the high runtimes for VPR. Second, SAT solvers have difficulties in proving the unsatisfiability of multiplier miters which is reflected by the PicoSAT runtimes. In fact, in our experiments PicoSAT aborted the computation due to a lack of memory for multipliers with $n = 16$ and higher. Consequently, we could not conduct measurements for the consumer side of the tool flow for these functions. We mark these cases with an asterisk in Table 2.

Table 3: Runtimes comparison between consumer and producer.

Test function	Consumer	Producer	
	Total [s]	Total [s]	workload
Converter	0.165	2.668	93%
128-bit parity	0.190	11.683	98%
8-bit add/sub	0.215	2.638	92%
16-bit add/sub	0.245	6.054	96%
32-bit add/sub	0.283	15.209	98%
64-bit add/sub	0.375	42.092	99%
6-bit multiplier	0.726	6.508	89%
8-bit multiplier	11.515	30.334	72%
10-bit multiplier	192.651	302.339	61%

The main observation from Table 2 is the difference in time effort between producer and consumer. DeComPose is less costly than ComPose, so is ABC$'$ in comparison to ABC with the gap widening for the more complex test functions. Importantly, with one exception there is a notable difference between PicoSAT and TraceCheck runtimes, even if not as pronounced as expected. This might be due to the fact that on one hand unsatisfiability is easily proven for the miters built from our simple test functions, and on the other hand the generated netlists are quite large which means that all tools processing netlists spend substantial time in file I/O. For the more complex test functions the SAT solver dominates the producer's overall runtime, an effect that can be seen in going from the 8-bit to the 10-bit multiplier.

With Table 3, we give an overview of the total runtimes for both the consumer and producer side. The table also reports the producer's percentage of the total workload consisting of both the consumer's and producer's runtime. The data shows that we succeeded in our plan to shift the majority of the workload from the consumer platform to an external resource.

To further underline this key point, we conducted runtime measurements with cnf problems from the 2008 SAT-Race_TS_1 benchmark suite. Table 4 compares the runtimes of the SAT solver (PicoSAT) and the proof checker (TraceCheck) for a number of benchmark problems. The cnfs differ greatly in the number of variables and clauses but include instances with up to six orders of magnitude more variables and clauses (`narai_vpn-10 s`) than the miters for our test functions. The last column of Table 4 shows that the effort for checking a proof trace is between one and three (five in an extreme case) orders of magnitude lower than for proving equivalence and generating the proof trace.

Table 5 gives the results for the memory usage for each tool and test function. We measured the peak memory usage with Valgrind [31], that is, the massif tool, and included stacks as well as heaps in the measurements. The first set of columns displays the memory usage on producer side, and columns eight to ten show the measurement results on

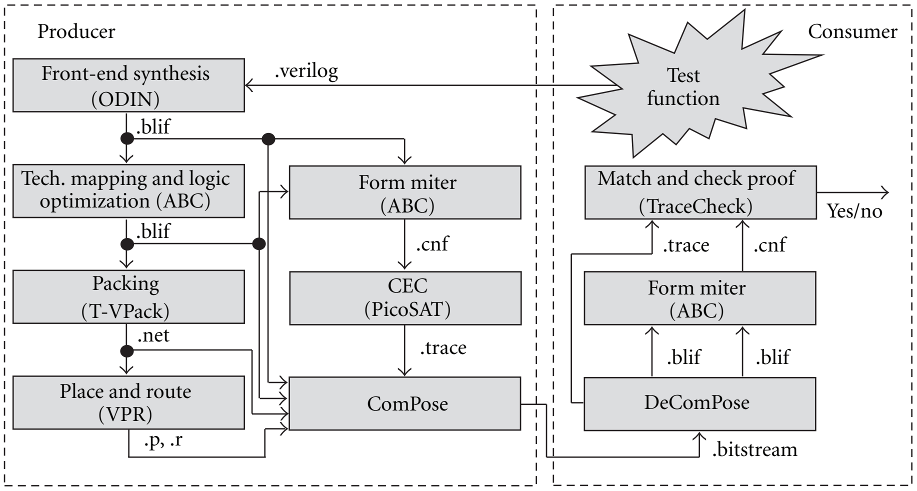

FIGURE 4: Runtime cec prototype tool flow.

TABLE 4: Runtime comparison between PicoSAT (SAT) and TraceCheck (Check) for benchmarks of the 2008 SAT-Race_TS_1.

cnf instance	Size of cnf [Vars × Clauses]	SAT [s]	Check [s]	Factor
een-tip-sr06-par1	163647 × 484827	6.412	0.024	267
een-tipb-sr06-tc6b	40196 × 115775	3.028	0.012	252
godlb-heqc-desmul	28902 × 179895	75.697	1.512	50
goldb-heqc-rotmul	5980 × 35229	31.458	2.272	13
hooons-vbmc-s04-05	8503 × 25097	11.065	1.536	7
hooons-vbmc-s04-07	25900 × 77627	158.294	8.557	18
manol-pipe-c10b	43517 × 129265	87.937	1.075	81
manol-pipe-c10ni_s	204664 × 609478	255.584	0.004	63896
manol-pipe-c6id	82022 × 242044	5.460	0.052	105
manol-pipe-c6n	37147 × 110077	49.547	0.820	60
manol-pipe-c6nid_s	148051 × 438562	5.528	0.020	276
manol-pipe-c7_i	13023 × 38509	17.685	0.292	60
manol-pipe-c7idw	112620 × 333058	131.444	1.412	93
manol-pipe-c8_i	14052 × 41596	74.433	2.028	36
manol-pipe-c8b_i	32057 × 95005	13.485	0.256	52
manol-pipe-c8n	53697 × 159595	111.351	1.988	56
manol-pipe-f6b	37002 × 109570	6.672	0.328	20
manol-pipe-f6n	37452 × 110920	7.500	0.236	31
manol-pipe-g10idw	174122 × 516784	141.241	1.964	71
manol-pipe-g6bid	40371 × 118192	5.272	0.112	47
manol-pipe-g7n	23936 × 70492	5.784	0.248	23
narai_vpn-10s	2270930 × 8901946	306.335	0.368	832
schup-l2s-s04-abp4	14809 × 48429	67.528	7.896	8
velev-npe-1.0-02	3295 × 35407	23.577	3.748	6
velev-sss-1.0	1453 × 12526	20.741	3.744	5

consumer side. As in Table 2, the numbers marked with an asterisk represent a minimum value, measured before the tool aborted the computation as it ran out of memory.

We note that PicoSAT and TraceCheck, followed by VPR and ABC, are the most memory consuming tools on the producer's and consumer's side, respectively. Formal verification is therefore among the most memory demanding tools in the scenario. The larger multiplier test functions with 16-bit inputs or more caused PicoSAT to run out of memory. Comparing TraceCheck and PicoSAT, the consumer has to invest slightly more in memory than the producer. As opposed to the runtimes where the producer is able to

Table 5: Memory usage measurements for producer and consumer in the online CEC scenario.

Test function	Producer					Consumer			
	Odin	ABC	T-VPack	VPR	PicoSAT	ComPose	DeComPose	ABC$'$	TraceCheck
Converter	0.997 MiB	13.79 MiB	239.6 KiB	1.207 MiB	69.42 KiB	62.17 KiB	62.12 KiB	10.06 MiB	81.00 KiB
128-bit parity	935.8 KiB	13.60 MiB	310.1 KiB	3.465 MiB	189.2 KiB	61.78 KiB	61.73 KiB	9.937 MiB	181.1 KiB
8-bit add/sub	711.4 KiB	13.90 MiB	243.5 KiB	1.218 MiB	87.71 KiB	61.62 KiB	61.58 KiB	10.54 MiB	101.1 KiB
16-bit add/sub	771.4 KiB	14.05 MiB	267.4 KiB	2.306 MiB	238.8 KiB	61.62 KiB	61.58 KiB	10.70 MiB	268.4 KiB
32-bit add/sub	899.6 KiB	14.27 MiB	315.4 KiB	4.480 MiB	488.5 KiB	61.62 KiB	61.58 KiB	10.98 MiB	485.9 KiB
64-bit add/sub	1.130 MiB	14.58 MiB	533.8 KiB	8.341 MiB	1.010 MiB	61.62 KiB	61.58 KiB	11.69 MiB	1.038 KiB
6-bit multiplier	774.6 KiB	13.93 MiB	263.6 KiB	2.298 MiB	1.380 MiB	62.70 KiB	62.66 KiB	10.38 MiB	2.167 MiB
8-bit multiplier	803.9 KiB	13.95 MiB	301.1 KiB	4.086 MiB	25.74 MiB	64.28 KiB	64.23 KiB	10.57 MiB	43.97 MiB
10-bit multiplier	887.3 KiB	14.10 MiB	391.9 KiB	6.425 MiB	374.8 MiB	71.20 KiB	71.15 KiB	10.87 MiB	652.8 MiB
16-bit multiplier	1.201 MiB	14.42 MiB	937.0 KiB	16.46 MiB	2.477 GiB*				
32-bit multiplier	2.750 MiB	19.83 MiB	3.590 MiB	67.33 MiB	2.742 GiB*				
64-bit multiplier	8.858 MiB	48.98 MiB	14.41 MiB	271.0 MiB	2.699 GiB*				

Table 6: Size measurements for the bitstream and proof trace.

Test function	Bitstream size [KiB]	Proof trace size [KiB]	Overhead [%]
Converter	120.2	23.2	19.35
128-bit parity	309.8	61.2	19.78
8-bit add/sub	99.4	32.8	32.98
16-bit add/sub	246.9	103.9	42.10
32-bit add/sub	490.6	194.3	39.60
64-bit add/sub	1077.9	464.9	43.13
6-bit multiplier	1657.6	1519.0	91.64
8-bit multiplier	40803.3	40538.6	99.35
10-bit multiplier	711487.3	711052.8	99.93

burden a major part of the total workload, the memory resource requirement is still considerable for the consumer in our current prototype tool flow.

We further provide an estimate on the overhead incurred by adding a proof trace to the delivered bitstream. The accuracy of the estimate is limited due to two facts. First, we do not synthesize for an FPGA of given size but operate VPR in the area minimization mode. For each circuit, VPR chooses a logic block array just large enough to accommodate the circuit and a channel width just sufficient to route the circuit. Consequently, our overheads are relative to the circuit size which results in a slightly pessimistic estimate. Second, our tool flow does not yet specify a binary bitstream format as ComPose merely forms a concatenation of several text files. Hence, we measure the size of the prototype bitstream which is composed of the corresponding text files, being a text file itself. Table 6 lists for each test function the size of the bitstreams as created by ComPose, the proof trace as calculated by PicoSAT, and the corresponding overhead, that is, the percentage of the bitstream made up by the resolution proof trace. The results show a wide variation in overheads from the rather low 19% for the EBCDIC-ASCII converter to 99% for the 10-bit multiplier. Although these numbers are overly pessimistic and coarse estimates, they point to the need of investigating a binary format for the proof-carrying bitstream using compression techniques.

6. Conclusion and Future Work

In this paper, we present proof-carrying hardware (PCH) as a novel approach to reconfigurable system security. We describe the main concept of PCH and detail its advantage over known reconfigurable hardware security approaches. Then, we apply the PCH concept to runtime combinational equivalence checking (CEC). CEC guarantees the adherence to functional specifications of a module which eliminates a major security risk for the consumer when loading reconfigurable modules at runtime. We elaborate on a prototype tool flow for runtime CEC, show its implementation, and discuss experimental results. Using proof traces generated by a modern SAT solver, the prototype clearly indicates a significant shift of computational cost and memory usage from the reconfigurable consumer platform to the producer of a reconfigurable hardware module.

While the experiments in this paper already demonstrate the high potential of the PCH approach, we will complete the tool flow shown in Figure 3 as a next step. We will extend the tool flow to formally verify all production steps and finalize all required tools on the consumer side. Future work also includes the definition of a binary proof-carrying bitstream format and the investigation of compression techniques, which will allow us to demonstrate a complete PCH framework.

Furthermore, we are interested in applying PCH to safety properties other than combinational equivalence. The range of safety policies to be investigated includes the following.

(i) First is the sequential equivalence of modules based on ABC which is capable of building miter functions not only for combinational but also for sequential circuits using and-inverter graphs as described in [32]. Alternatively, we intend to look into the VIS tool that uses BDDs to perform sequential equivalence checks; see [33].

(ii) Second is the absence of short circuits which is another functional property. As [34] has shown the partial reconfiguration process can result in short-term, middle-term, or long-term short circuits damaging the system in different ways.

(iii) Third is the physical (structural) isolation through primitives such as moats and drawbridges introduced in [11]. Here, we plan to extend VPR to enforce such primitives during place and route and annotate the bitstream accordingly to allow the consumer to efficiently verify isolation.

(iv) Fourth are the other nonfunctional properties such as specific timing and routing constraints which might be more efficiently guaranteed when the producer annotates the bitstream properly.

This overview indicates the variety of safety concerns and possible safety policies to which the PCH concept might be applicable. It is important to note that the PCH concept is broad enough to include different methods for verifying safety properties, ranging from classic formal verification techniques to checking annotated bitstreams. The main characteristics of PCH is the shift of work from the consumer to the producer which enables embedded target systems with limited resources to obtain security guarantees at runtime.

Acknowledgment

This work is partially funded by the International Graduate School Dynamic Intelligent Systems, University of Paderborn, Germany.

References

[1] G. Necula and P. Lee, “Proof-carrying code,” Tech. Rep. 15213, School of ComputerScience, Carnegie Mellon University, Pittsburgh, Pa, USA, CMU-CS-96-165, November 1996.

[2] R. Kastner and T. Huffmire, “Threats and challenges in reconfigurable hardware security,” in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA ’08)*, pp. 334–345, CSREA Press, July 2008.

[3] S. Drzevitzky, U. Kastens, and M. Platzner, “Proof-carrying hardware: towards runtime verification of reconfigurable modules,” in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig ’09)*, pp. 189–194, IEEE, Cancun, Mexico, December 2009.

[4] S. Drimer, “Volatile FPGA design security—a survey,” December 2007.

[5] T. Huffmire, C. Irvine, T. D. Nguyen, T. Levin, R. Kastner, and T. Sherwood, *Handbook of FPGA Design Security*, Springer, New York, NY, USA, 1st edition, 2010.

[6] R. Chaves, G. Kuzmanov, and L. Sousa, “On-the-fly attestation of reconfigurable hardware,” in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL ’08)*, pp. 71–76, IEEE, September 2008.

[7] S. Drimer and M. Kuhn, “A protocol for secure remote updates of FPGA configurations,” in *Reconfigurable Computing: Architectures,Tools and Applications*, vol. 5453, pp. 50–61, Springer, New York, NY, USA, 2009.

[8] S. Drimer, “Authentication of FPGA bitstreams: why and how,” in *Reconfigurable Computing: Architectures,Tools and Applications*, vol. 4419, pp. 73–84, Springer, New York, NY, USA, 2007.

[9] B. Badrignans, R. Elbaz, and L. Torres, “Secure FPGA configuration architecture preventing system downgrade,” in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL ’08)*, pp. 317–322, IEEE, September 2008.

[10] T. Huffmire, S. Prasad, T. Sherwood, and R. Kastner, “Policy-driven memory protection for reconfigurable hardware,” in *Proceedings of the European Symposium on Research in Computer Security (ESORICS ’06)*, vol. 4189 of *LNCS*, pp. 461–478, Springer, September 2006.

[11] T. Huffmire, B. Brotherton, G. Wang et al., “Moats and drawbridges: an isolation primitive for reconfigurable hardware based systems,” in *Proceedings of the Symposium on Security and Privacy*, pp. 281–295, IEEE, Oakland, Calif, USA, May 2007.

[12] T. Huffmire, T. Sherwood, R. Kastner, and T. Levin, “Enforcing memory policy specifications in reconfigurable hardware,” *Computers and Security*, vol. 27, no. 5-6, pp. 197–215, 2008.

[13] T. Huffmire, B. Brotherton, N. Callegari et al., “Designing secure systems on reconfigurable hardware,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 1–24, 2008.

[14] T. Huffmire, B. Brotherton, T. Sherwood et al., “Managing security in FPGA-based embedded systems,” *IEEE Design and Test of Computers*, vol. 25, no. 6, pp. 590–598, 2008.

[15] V. D’Silva, D. Kroening, and G. Weissenbacher, “A survey of automated techniques for formal software verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, Article ID 4544862, pp. 1165–1178, 2008.

[16] S. Singh and C. J. Lillieroth, “Formal verification of reconfigurable cores,” in *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCMM ’99)*, pp. 25–33, IEEE, April 1999.

[17] N. Necula and P. Lee, “Safe, untrusted agents using proof-carrying code,” in *Mobile Agents and Security*, vol. 1419 of *LNCS*, pp. 61–91, Springer, New York, NY, USA, 1998.

[18] K. Klohs and U. Kastens, “Memory requirements of java bytecode verification on limited devices,” in *Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification(COCV ’04)*, vol. 132, 2004.

[19] K. Klohs, “A summary function model for the validation of interprocedural analysis results,” in *Proceedings of the 7th International Workshop on Compiler Optimization meets Compiler Verification (COCV ’08)*, 2008.

[20] S. Myagmar, A. Lee, and W. Yurcik, “Threat modeling as a basis for security requirements,” in *Proceedings of the IEEE Symposium on Requirements Engineering for Information Security (SREIS ’05)*, August 2005.

[21] A. Mishchenko, S. Chatterjee, and R. Brayton, “DAG-aware AIG rewriting: a fresh look at combinational logic synthesis,” in *Proceedings of the Design Automation Conference (DAC ’06)*, pp. 532–535, ACM, July 2006.

[22] S. Chatterjee, A. Mishchenko, R. Brayton, and A. Kuehlmann, “On resolution proofs for combinational equivalence,” in *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC ’07)*, pp. 600–605, June 2007.

[23] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, “Improvements to combinational equivalence checking,” in *Proceedings of the IEEE/ACM International Conference on*

Computer-Aided Design (ICCAD '06), pp. 836–843, ACM, New York, NY, USA, 2006.

[24] "Odin A Verilog RTL Synthesis Tool for Heterogeneous FPGAs," http://www.eecg.toronto.edu/~jayar/software/odin/index.html.

[25] P. Jamieson and J. Rose, "A verilog RTL synthesis tool for heterogeneous FPGAS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 305–310, IEEE, Tampere, Finland, August 2005.

[26] "Going Places with ABC, Berkeley Logic Synthesis and Verification Group," http://www.eecs.berkeley.edu/~alanmi/abc/abc%20tutorial.ppt.

[27] V. Betz and J. Rose, "VPR: a new packing, placement and routing tool for FPGA research," in *Proceedings of the nternational Conference on Field Programmable Logic and Applications (FPL '97)*, vol. 1304, pp. 213–222, Springer, London, UK, 1997.

[28] V. Betz, "VPR and T-VPack User's Manual," (Version 5.0), 2008.

[29] A. Biere, "PicoSAT essentials," *Journal on Satisfiability , Boolean Modeling and Computation*, vol. 4, pp. 75–97, 2008.

[30] C. Sinz and A. Biere, "Extended resolution proofs for conjoining BDDs," in *Proceedings of the 1st International Computer Science Symposium in Russia (CSR '06)*, vol. 3967 of *LNCS*, pp. 600–611, Springer, 2006.

[31] J. Seward, N. Nethercote, and T. Hughes, "Valgrind Documentation," August 2009, http://valgrind.org/.

[32] R. Brayton and A. Mishchenko, "Scalably-verifiable sequential synthesis," ERl Technical Report, EECS Dept. UC Berkeley, 2007.

[33] T. V. Group, "Vis: a system for verification and synthesis," in *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)*, R. Alur and T. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, pp. 428–432, Springer, July 1996.

[34] D. Beckhoff, C. Koch, and J. Torresen, "Short-circuits on fpgas caused by partial runtime reconfiguration," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 596–601, IEEE, August 2010.