Formal Methods in Computer-Aided Design 2021

Scaling Up Hardware Accelerator Verification using A-QED with Functional Decomposition

Saranyu Chattopadhyay[*], Florian Lonsing[*], Luca Piccolboni[†], Deepraj Soni[¶], Peng Wei[§], Xiaofan Zhang[||], Yuan Zhou[‡], Luca Carloni[†], Deming Chen[||], Jason Cong[§], Ramesh Karri[¶], Zhiru Zhang[‡], Caroline Trippel[*], Clark Barrett[*], Subhasish Mitra[*]
[*]Stanford University, [†]Columbia University, [‡]Cornell University, [§]University of California, Los Angeles,
[¶]New York University, [||]University of Illinois, Urbana-Champaign

***Abstract*—Hardware accelerators (*HAs*) are essential building blocks for fast and energy-efficient computing systems. *Accelerator Quick Error Detection (A-QED)* is a recent formal technique which uses Bounded Model Checking for pre-silicon verification of HAs. A-QED checks an HA for *self-consistency*, i.e., whether identical inputs within a sequence of operations always produce the same output. Under modest assumptions, A-QED is both sound and complete. However, as is well-known, large design sizes significantly limit the scalability of formal verification, including A-QED. We overcome this scalability challenge through a new decomposition technique for A-QED, called *A-QED with Decomposition (A-QED^2)*. A-QED^2 systematically decomposes an HA into smaller, functional sub-modules, called *sub-accelerators*, which are then verified independently using A-QED. We prove completeness of A-QED^2; in particular, if the full HA under verification contains a bug, then A-QED^2 ensures detection of that bug during A-QED verification of the corresponding sub-accelerators. Results on over 100 (buggy) versions of a wide variety of HAs with millions of logic gates demonstrate the effectiveness and practicality of A-QED^2.**

I. Introduction

Hardware accelerators (*HAs*) are critical building blocks of energy-efficient System-on-Chip (*SoC*) platforms [1]–[3]. Unlike general-purpose processors, HAs implement a set of domain-specific functions (e.g., encryption, 3D Rendering, deep learning inference), referred to as *actions* in this paper, for improved energy and throughput. Today's SoCs integrate dozens of diverse HAs (e.g., 40+ HAs in Apple's A12 mobile SoC [4]).

Unfortunately, the energy and throughput improvements enabled by HAs come at the cost of increased design complexity. Ensuring that a given SoC will behave correctly and reliably requires verifying each and every constituent HA. Furthermore, HAs must achieve short design-to-deployment timelines in order to meet the needs of a wide variety of evolving applications [5]. Using conventional formal verification techniques to verify HAs faces several key challenges. Manually crafting extensive design-specific formal properties or full abstract functional specifications can be time-consuming and error-prone [6], [7]. Moreover, scaling verification to large HAs (with millions of logic gates) is difficult or even infeasible using off-the-shelf formal tools.

A recent formal verification technique targeting HAs, *Accelerator-Quick Error Detection (A-QED)* [8], overcomes the first challenge above. A-QED is readily applicable for a popular class of HAs: *loosely-coupled accelerators* (*LCAs*) [9], [10] (i.e., HAs that are not integrated as part of a central processing unit (*CPU*), but via an SoC's network-on-chip or a bus) that are also *non-interfering*. Non-interfering HAs produce the same result for a given action independent of their context within a sequence of actions (not to be confused with combinational circuits). In other words, the state of the accelerator does not affect future computations, and each computation is independent from previous computations. In contrast, computations of *interfering* HAs depend on state that is the result of previous computations. A-QED uses Bounded Model Checking (*BMC*) [11] to symbolically check sequences of actions for *self-consistency*. Specifically, it checks for *functional consistency (FC)*, the property that identical inputs within a sequence of operations always produce the same outputs. It was shown that FC checks, together with *response bound (RB)* checks and *single-action correctness (SAC)* checks, provide a thorough verification technique for non-interfering LCAs [8]. However, despite its success in discovering bugs in moderately-sized HA designs, A-QED suffers from the scalability challenges of formal tools. For example, A-QED (backed by off-the-shelf formal verification tools) times out after 12 hours when run on NVDLA, NVIDIA's deep-learning HA [12] with approximately 16 million logic gates.

In this paper, we present a new verification approach called *A-QED with Decomposition (A-QED^2)* to address the scalability challenge. First, we introduce a new, more general formal model of HA execution, which captures both interfering and non-interfering LCAs. We then show how A-QED^2 can *decompose* a large LCA into smaller *sub-accelerators* in such a way that both FC and RB checks can be directly applied to the sub-accelerators. Unlike conventional verification approaches based on decomposition, no new properties need to be devised to apply FC and RB to the decomposed sub-accelerators. Existing decomposition approaches can be leveraged to additionally check SAC of the sub-accelerators. A-QED^2 is complementary to verification approaches that rely on design abstraction, which can be used to further improve scalability and to simplify the effort required for SAC checks on decomposed sub-accelerators.

This paper presents both a formal foundation of A-QED^2 and an empirical evaluation that demonstrates its bug-finding capabilities in practice. We prove that A-QED's completeness guarantees [8] continue to hold for A-QED^2—if the full HA

https://doi.org/

under verification contains a bug, then A-QED^2 will detect that bug. Furthermore, we apply A-QED^2 to a wide variety of non-interfering LCAs (although our theoretical proofs apply to interfering LCAs as well): 109 different (buggy) versions of large open-source HAs of up to 200 million logic gates (including industrial HAs). Our empirical results focus on designs which are described in a high-level language (e.g., C/C++) and then translated to Register-Transfer-Level (*RTL*) designs (e.g., Verilog) using High-Level Synthesis (*HLS*) flows, where appropriate optimizations like pipelining and parallelism are instantiated. Such HLS-based HA design flows are becoming increasingly common in industry. However, A-QED^2 is not restricted to these specific HA design styles. Our empirical results show:

1) Off-the-shelf formal tools cannot handle large HAs with millions of logic gates, even when the HAs are expressed as high-level C/C++ designs. In our experiments, A-QED verification of many such HAs times out after 12 hours or runs out of memory.
2) A-QED^2 is broadly applicable to a wide variety of HAs and detects all bugs detected by conventional simulation-based verification. For very large HAs with several million (up to over 200 million) logic gates, A-QED^2 detects bugs in less than 30 minutes in the worst case and in a few seconds in most cases.
3) A-QED^2 is thorough – it detected all bugs that were detected by conventional (simulation-based) verification techniques. At the same time, A-QED^2 improves verification effort significantly compared to simulation-based verification – $\sim 5X$ improvement on average, with $\sim 9X$ improvement (one person month with A-QED^2 vs. 9 person months with conventional verification flows) for the large, industrial designs.

The rest of this paper is organized as follows. Sec. II presents related work. Sec. III presents a formal model of the accelerators targeted by A-QED^2 and our decomposition technique. Sec. IV details the A-QED^2 algorithms. Results are presented in Sec. V, and Sec. VI concludes.

II. Related Work

Conventional formal HA verification, e.g., [13]–[16], requires a specification, typically in the form of manually written, design-specific properties. These are then combined with a formal model of the design and handed to a formal tool, which attempts to prove the properties or find counter-examples. For the verification of latency-insensitive designs, an approach was developed to automatically derive and check properties from the RTL synthesized in HLS flows [17]. However, these derived properties are targeted at specific types of bugs.

Large design sizes have always been a challenge for formal techniques, and various approaches to this problem have been proposed. Among techniques to improve scalability are abstraction [18] and compositional reasoning (cf. [19]). The former removes details of the design, gaining scalability at the cost of possible false errors. Finding a scalable abstraction that does not generate false errors can be difficult and may be impossible in some cases. The latter uses *assume-guarantee* reasoning (e.g., [20]–[25]) and can be applied to decompose a large HA into smaller sub-modules. Importantly, the property p of the HA to be verified must also be decomposed into properties of the sub-modules. The properties of the sub-modules are verified individually under certain assumptions about the behavior of the other sub-modules. If all the properties of the sub-modules hold under the respective assumptions, then it can be concluded that p holds. However, finding the right properties for this decomposition can be very challenging.

Unlike for general compositional reasoning, the two main components of A-QED^2 (FC and RB) do not require decomposing properties. FC, in particular, leverages a universal *self-consistency* property. Self-consistency expresses the property that a design is expected to produce the same outputs whenever it is provided with the same inputs [26]. In A-QED^2, self-consistency is checked independently for each sub-module (sub-accelerator in our case). Importantly, these aspects of A-QED^2 do not require complex assumptions about the behavior of the other sub-modules.

It is challenging to establish general *completeness guarantees* for conventional formal verification techniques [27]–[31], since completeness depends on the set of properties being checked. Designer-guided approaches [32], [33] require manual effort. Automatic generation of properties is usually incomplete and depends on abstract design descriptions [34] or models [35], or analysis of simulation traces [36], which may be difficult. In contrast, we have general completeness results for A-QED^2.

A-QED^2 builds on A-QED [8] and leverages BMC [11], [37]. Similar approaches based on self-consistency have been successfully applied to other classes of hardware designs, such as processor verification (as *symbolic quick error detection (SQED)* [38]–[43]), as well as to hardware security [44]–[49].

III. Formal Model and Theoretical Results

In this section, we introduce a formal model for HAs, define functional consistency (*FC*), single-action correctness (*SAC*), and responsiveness for the model, and show how these properties provide correctness guarantees. We then define a notion of functional composition for our model and show how the above properties can be applied in a compositional way.

Our formal model differs from the one in previous work [8] in several important ways. It allows multiple inputs to be provided simultaneously by explicitly modeling the notion of *input batches*. The HAs we consider are *batch-mode accelerators* as they process input batches and produce output batches. Modeling batches is useful because it more closely matches the interfaces of real HAs. Moreover, input batches enable *intra-batch checks* for FC checking, as we describe below. With intra-batch checks, only one input batch is used for FC checking. Intra-batch checks are more restricted than general FC checks. However, they are easier to set up and run in practice, and they are highly effective at finding bugs, as we demonstrate empirically.

Our model also explicitly separates control states and memory states. Control states represent control-flow information such as, e.g., program counters in HLS models of HAs. Memory states represent all other state-holding elements, e.g., program variables.

In our model we distinguish starting and ending control states in which inputs are provided and the computed outputs are ready, respectively. This makes the formulation simpler and is also a better match for HLS designs written in a high-level language, which is our main target in the experimental evaluation. Further, our model enables us to formulate the notion of *strong FC*, which leads to a complete approach to bug-finding with only two input batches.

In previous work [8], a ready-valid protocol was used to model input/output transactions in RTL designs. In contrast, our focus is on HLS designs. Finally, we distinguish so-called *relevant states*, which are parts of the state space that can affect output values. This makes it possible to model interfering as well as non-interfering HAs. In our experiments we focus on non-interfering HAs.

Before presenting formal definitions, we illustrate terminology informally with an example of a non-interfering batch-mode HA as shown in Listing 1 (a slightly modified excerpt of an HA implementing AES encryption [50]).

Function `fun` of the HA has two sub-accelerators in lines 8-10 and 13-14 which are identified and verified by A-QED^2. Each sub-accelerator applies a certain operation to all inputs in an input batch of HA. In general, the *batch size* of an HA is the number of inputs in each batch, which is 256 for this HA. The first sub-accelerator *ACC₁* processes an input batch provided via `data` and stores its output batch in `buf`. The second sub-accelerator ACC_2 takes its input batch from `buf`, where it also stores the output batch it produces. The control state of the HA is only implicitly represented by the program counter when executing function `fun`. Variables `key` and `local_key` are global and determine the relevant state of the HA on which the result of the encryption operation depends. The HA is non-interfering because `key` and `local_key` are left unchanged by ACC_1 and ACC_2. Constants `BS`, `UF`, and `US` are used in HLS to configure the generated RTL.

Listing 1: HA Example (AES Encryption)

```
1  #define BS ((1) << 12) // BUF_SIZE
2  #define UF 2 // UNROLL_FACTOR
3  #define US BS/UF // UNROLL_SIZE
4
5  void fun(int data[BS], int buf[UF][US], int key[2]){
6    int j, k;
7    // ===ACC1 START===
8    for(j=0; j<UF; j++)
9      for(k = 0; k < BS/UF; k ++)
10       buf[j][k] = *(data + i*BS + j*US + k)^key[0];
11   // ===ACC1 END===
12   // ===ACC2 START===
13     for(j=0; j<UF; j++){
14       aes256_encrypt(local_key[j], buf[j]);}
15   // ===ACC2 END===
16 }
```

Definition 1. *A* batch-mode hardware accelerator (HA) *is a finite state transition system [51], [52]* $Acc := (b, A, D, O, S, s_{c,I}, s_{c,F}, S_{m,I}, T)$, *where*

- $b \in \mathbb{N}$ *with* $b \geq 1$ *is the* batch size,
- A *is a finite set of actions,*
- D *is a finite set of data values,*
- O *is a finite set of outputs,*
- $S = S_C \times S_M$ *is the set of* states *consisting of* control states S_C *and* memory states $S_M = S_{In} \times S_{Out} \times S_R \times S_N$, *where*
  - $S_{In} = (A \times D)^b$ *are the* input states,
  - $S_{Out} = O^b$ *are the* output states,
  - S_R *are the* relevant states, *and*
  - S_N *are the* non-relevant states,
- $s_{c,I} \in S_C$ *is the unique* initial control state, *which defines the set* $S_I = \{s_{c,I}\} \times S_M$ *of* initial states,
- $s_{c,F} \in S_C$ *is the unique* final control state, *which defines the set* $S_F = \{s_{c,F}\} \times S_M$ *of* final states,
- $S_{m,I}$ *is the set of allowable initial memory states, which defines the set* $S_{CI} = \{s_{c,I}\} \times S_{m,I}$ *of* concrete initial states,
- *and* $T : S \to S$ *is the* state transition function.

When referring to different HAs, e.g., Acc_0 and Acc_1, we use subscript notation to identify their components, e.g., $Acc_0 := (b_0, A_0, D_0, O_0, S_0, s_{c,I,0}, s_{c,F,0}, S_{m,I,0}, T_0)$.

We use $\boldsymbol{v} = \langle v_1, \ldots, v_{|\boldsymbol{v}|} \rangle$ to denote a sequence with elements denoted v_i and length $|\boldsymbol{v}|$. We concatenate sequences (and for simplicity of notation, single elements with sequences) using '$\cdot$', e.g., $\boldsymbol{v} = v_1 \cdot \boldsymbol{v'}$, where $\boldsymbol{v'} = \langle v_2, \ldots, v_{|\boldsymbol{v}|} \rangle$. We will sometimes identify a sequence $\boldsymbol{v}$ with the corresponding tuple, and we write $v \in \boldsymbol{v}$ to denote that v appears in $\boldsymbol{v}$. We denote the i-th element of a tuple t as $t(i)$.

An HA Acc operates on a set I^b of *input batches*, where b is the *batch size* and $I = A \times D$. An input batch $in \in I^b$ has b *batch elements*, each consisting of a pair (a, d) containing an action $a \in A$ to be executed and data $d \in D$ (the data on which action a operates).

A state $s \in S$ of Acc with $s = (s_c, s_m)$ consists of a control state $s_c \in S_C$ and a memory state $s_m \in S_M$. The control state s_c represents control-flow-related state (e.g., the program counter in an execution of a high-level model of Acc). In a run of Acc, the control state starts at a distinguished initial state $s_{c,I}$ and ends at a distinguished final state $s_{c,F}$.

The memory state represents all other state-holding elements of Acc (including, e.g., global variables, local variables, function parameters, and memory elements). The memory state $s_m = (s_{in}, s_{out}, s_r, s_n)$ is divided into four parts. The first part, $s_{in} \in S_{In}$, contains the input to Acc. More precisely, in a run of Acc, the value of s_{in} in the initial state is considered the input for that run. Similarly, at the end of a run of Acc, $s_{out} \in S_{Out}$ contains the outputs for that run (i.e., the values computed by Acc based on the inputs present at the start of the run).

The relevant state s_r represents those state elements (other than s_{in}) that can influence the values of the outputs. Any part of the state that can affect the output value in at least one execution should be included in the relevant state. As an example of when this is needed, consider an encryption HA with actions for setting the encryption key and for encrypting data. The internal state that stores the key is part of the relevant state because it affects the way the output is computed from the input. The non-relevant state s_n is everything else. We write $\mathit{ctrl}(s)$, $\mathit{mem}(s)$, $\mathit{inp}(s)$, $\mathit{out}(s)$, $\mathit{rel}(s)$, and $\mathit{nrel}(s)$ to denote the components s_c, s_m, s_{in}, s_{out}, s_r, and s_n, respectively. We overload the latter four operators to apply to memory states as well, and we lift the notation to sequences of states.

The set S_I of initial states contains all states resulting from combining a memory state in S_M with the unique initial control state $s_{c,I}$. The concrete initial states, S_{CI}, are a subset of S_I, and essentially represent the reset state(s) of the HA. They play a role in defining the *reachable* states (see Definition 3, below). The set S_F of final states contains all states resulting from combining a memory state in S_M with the unique final control state $s_{c,F}$. Finally, the transition function T defines the successor state for any given state in S.

Given an input batch $\mathit{in} \in I^b$, the HA produces an *output batch* $o \in O^b$ as follows. Let $s_0 \in S_I$ be an initial state with $\mathit{inp}(s_0) = \mathit{in}$, and let $\boldsymbol{s} = \boldsymbol{T}(s_0) = \langle s_1, \ldots, s_k \rangle$ denote the sequence of $|\boldsymbol{s}| = k$ *successor states* generated by the *transition function* T, where $s_i = T(s_{i-1})$ for $1 \leq i \leq k$, such that $s_k \in S_F$ is a final state (and no earlier states in $\boldsymbol{s}$ are final states). We also assume, without loss of generality, that $\mathit{ctrl}(s_i) \neq s_{c,I}$ for $i > 0$. The final state s_k holds the output batch $\mathit{out}(s_k) = o$ with $o \in O^b$ that is produced for the input batch $\mathit{inp}(s_0) = \mathit{in}$. Given a sequence $\boldsymbol{s}$, we write $\mathit{initsym}(\boldsymbol{s})$ and $\mathit{final}(\boldsymbol{s})$ to denote the subsequence of $\boldsymbol{s}$ containing all initial and final states that occur in $\boldsymbol{s}$, respectively.

Given a sequence of input batches, an HA generates a sequence of output batches based on concatenating executions for each input batch.

Definition 2. *Let $\boldsymbol{in}$ be a sequence of inputs with $n = |\boldsymbol{in}|$, and let $s_0 \in S_I$. Then, $\mathit{StateSeq}(\boldsymbol{in}, s_0)$ denotes the sequence of* successor states *of s_0 that result from executing $\boldsymbol{in}$, which is defined as follows.*

- *Let s'_0 be the result of replacing $\mathit{inp}(s_0)$ with in_1 in s_0. Let $\boldsymbol{s'} = s'_0 \cdot \boldsymbol{T}(s'_0)$.*
- *If $|\boldsymbol{in}| = 1$ then $\mathit{StateSeq}(\boldsymbol{in}, s_0) = \boldsymbol{s'}$*
- *If $|\boldsymbol{in}| > 1$, then*
  - *let $s_f = \mathit{final}(\boldsymbol{s'})$ (which is unique),*
  - *let $s_i = (s_{c,I}, \mathit{mem}(s_f))$,*
  - *let $\boldsymbol{s''} = \mathit{StateSeq}(\langle \mathit{in}_2, \ldots, \mathit{in}_n \rangle, s_i)$.*
  - *Then, $\mathit{StateSeq}(\boldsymbol{in}, s_0) = \boldsymbol{s'} \cdot \boldsymbol{s''}$.*

In Definition 2, the state s_i from which each subsequent input batch is executed is obtained from the final state s_f produced from executing the previous input batch. Given an HA Acc, we write $\mathit{StateSeq}(\mathit{Acc}, \boldsymbol{in}, s_0)$ to explicitly refer to the successor states of s_0 generated by Acc. If Acc is clear from the context, we omit it.

Definition 3. *A state $s \in S$ is* reachable *if $s \in S_{CI}$ or if there exists a* concrete initial state *$s_0 \in S_{CI}$ and sequence $\boldsymbol{in}$ of input batches such that $s \in \mathit{StateSeq}(\boldsymbol{in}, s_0)$. A relevant state s_r is reachable if $s_r = \mathit{rel}(s)$ for some reachable state s.*

Note that the initial states S_I are not necessarily all reachable.

Next, we define an abstract specification for an HA function. Note that we use this to define correctness, but one of the features of A-QED is that the specification is not needed for the main verification technique.

Definition 4 (Abstract Specification)**.** *For an HA Acc, let $\mathit{Spec} : I \times S_R \rightarrow O$ be an* abstract specification function*.*

Definition 4 states that the value of an output computed by an HA is completely determined by the corresponding input and the relevant part of the memory state when the HA was started. Note that the inclusion of the relevant memory state makes the definition general enough to model interfering HAs. To model non-interfering HAs, we can either make the output dependent on only the input batch, or require that the relevant state does not change in state transitions.

Based on the abstract specification, we define the *functional correctness* of an HA in terms of the output batches that are produced for given input batches as follows.

Definition 5 (Functional Correctness)**.** *An HA Acc is* functionally correct *with respect to an abstract specification Spec if, for all concrete initial states $s_0 \in S_{CI}$ and all sequences $\boldsymbol{in}$ of input batches, if*

- *$\boldsymbol{in} = \langle \mathit{in}_1, \ldots, \mathit{in}_n \rangle$,*
- *$\boldsymbol{s} = \mathit{StateSeq}(\boldsymbol{in}, s_0)$,*
- *$\boldsymbol{s_I} = \mathit{initsym}(\boldsymbol{s}) = \langle s_{I,1}, \ldots, s_{I,n} \rangle$,*
- *$\boldsymbol{o} = \mathit{out}(\mathit{final}(\boldsymbol{s})) = \langle o_1, \ldots, o_n \rangle$,*

then $\forall\, j \in [1 \ldots b].\ o_n(j) = \mathit{Spec}(\mathit{in}_n(j), \mathit{rel}(s_{I,n}))$.

A bug is simply a failure of functional correctness.

As mentioned above, even without a formal specification, we can apply the core technique of A-QED. To do so, we leverage the concept of *functional consistency*, the notion that under modest assumptions, two identical inputs will always produce the same outputs.

Definition 6 (Functional Consistency (FC))**.** *An HA Acc is* functionally consistent *if, for all concrete initial states $s_0 \in S_{CI}$ and for all sequences $\boldsymbol{in}$ of input batches, if*

- *$\boldsymbol{in} = \langle \mathit{in}_1, \ldots, \mathit{in}_n \rangle$, $\boldsymbol{s} = \mathit{StateSeq}(\boldsymbol{in}, s_0)$,*
- *$\boldsymbol{s_I} = \mathit{initsym}(\boldsymbol{s}) = \langle s_{I,1}, \ldots, s_{I,n} \rangle$,*
- *$\boldsymbol{o} = \mathit{out}(\mathit{final}(\boldsymbol{s})) = \langle o_1, \ldots, o_n \rangle$,*

then $\forall\, i \in [1, n], j, j' \in [1, b].$

$$\mathit{in}_i(j) = \mathit{in}_n(j') \wedge \mathit{rel}(s_{I,i}) = \mathit{rel}(s_{I,n}) \rightarrow o_i(j) = o_n(j').$$

Definition 6 illustrates the need for the *relevant* designation for memory states. It essentially says that two inputs, even if started at different times and in different batch positions, should produce the same output, as long as the relevant part of the memory is the same when the two inputs are sent in. The following lemma is straightforward (see the online appendix [53] for proofs of this and other results).

Lemma 1 (Soundness of FC)**.** *If an HA is functionally correct, then it is functionally consistent.*

Checking FC requires running BMC over multiple iterations of the HA and may be computationally prohibitive for large designs or for large values of n. Often, it is possible to verify a stronger property, which only requires checking consistency across two runs of the HA.

Definition 7 (Strong FC)**.** *An HA Acc is* strongly functionally consistent *if, for all reachable initial states s_0, s_0' and input batches in, in', if*

- $\boldsymbol{s} = \mathit{StateSeq}(\langle in \rangle, s_0)$, $\boldsymbol{s}' = \mathit{StateSeq}(\langle in' \rangle, s_0')$,
- $\boldsymbol{s_F} = \mathit{final}(\boldsymbol{s}) = \langle s_F \rangle$, $\boldsymbol{s_F}' = \mathit{final}(\boldsymbol{s}') = \langle s_F' \rangle$,
- $\boldsymbol{o} = \mathit{out}(\boldsymbol{s_F}) = \langle o \rangle$, $\boldsymbol{o}' = \mathit{out}(\boldsymbol{s_F}') = \langle o' \rangle$,

then $\forall\, j, j' \in [1, b]$.

$$in(j) = in'(j') \wedge \mathit{rel}(s_0) = \mathit{rel}(s_0') \rightarrow o(j) = o'(j').$$

The main difference between FC and strong FC is that the initial states s_0 and s_0' can be any reachable states. In contrast to that, the initial state $s_0 \in S_{CI}$ in the definition of FC is a concrete one. It is easy to see that strong FC implies FC, but the reverse is not true in general. This is because it may not be possible for two reachable initial states s_0 and s_0' chosen in a strong FC check to both appear in a single sequence of states resulting from executing a sequence of input batches starting in a concrete initial state. Similar to previous work on A-QED for non-batch-mode HAs [8], FC checking relies on sequences of input batches to reach all reachable states from a concrete initial state. For strong FC checking, on the other hand, two individual input batches are sufficient because the two initial states s_0 and s_0' can be arbitrarily chosen from the reachable states. Like FC, strong FC is a sound approach.

Lemma 2 (Soundness of Strong FC)**.** *If an HA is functionally correct then it is strongly functionally consistent.*

A challenge with using strong FC is that it requires starting with reachable initial states. However, we found that in practice (cf., Section V), it is seldom necessary to add any constraints on the initial states. This may seem surprising given the well-known problem of spurious counterexamples that arises when using formal to prove functional correctness without properly constraining initial states. There are at least two reasons for this. First, many HAs have less dependence on internal state (none for non-interfering HAs) than other kinds of designs. But second, and more importantly, FC is a much more forgiving property than design-specific correctness. Many designs are functionally consistent, even when run from unreachable states. In fact, we believe that this is a natural outcome of good design and that designing for FC is a sweet spot in the trade-off between design for verification and other design goals. If designers take care to ensure FC, even from unreachable states, then strong FC is both sound and easy to formulate.

Even simpler versions of the checks above can be obtained by making them *intra-batch* checks. An HA is *intra-batch functionally consistent* if it is functionally consistent when $i = n = 1$. That is, intra-batch FC checks are based on sending a single input batch to the HA. Consequently, it is not necessary to identify and compare the relevant parts of the initial states (cf. Definition 6) as there is precisely one initial state being used. Similarly, an HA is *intra-batch strongly functionally consistent* if it is strongly functionally consistent when $s_0 = s_0'$ and $in = in'$. Again, only one input batch is sent to the HA and the relevant parts of the initial states are thus always equal. As we will show in Section V, intra-batch checks can be a very effective approach for cheaply finding bugs. Intra-batch checks are applicable only to batch-mode HAs; i.e., they are not applicable in the context of A-QED targeted at HAs processing sequences of single inputs [8] rather than input batches.

While functional consistency alone can find many bugs, it becomes a complete technique (i.e., it finds all bugs) by combining it with *single-action checks*.

Definition 8 (Single-Action Correctness (*SAC*))**.** *An HA Acc is* single-action correct (*SAC*) *with respect to an abstract specification Spec if, for every batch element (a, d) and for every reachable relevant state s_r, there exists some reachable initial state s, such that $\mathit{inp}(s)(j) = (a, d)$ for some j, $\mathit{rel}(s) = s_r$, and $\mathit{out}(\mathit{final}(\boldsymbol{T}(s)))(j) = \mathit{Spec}((a, d), s_r)$.*

Essentially, SAC requires that for each action a, data d, and reachable relevant state s_r, we have checked that the result is computed correctly when starting from some reachable initial state s whose relevant state matches s_r. For every batch element (a, d) and s_r, it is sufficient to run a single check where we can choose (a, d) to be at any arbitrary position j in the batch $\mathit{inp}(s)$. Checking SAC *does* require using the specification explicitly, but these kinds of checks typically already exist in unit or regression tests. SAC may even be possible to verify using simulation. As we show in Section V, many bugs can be discovered without checking SAC at all.

When formalizing single-action checks, we again advocate using an over-approximation for reachability and encourage the design of HAs with simple over-approximations for the set of reachable relevant states. For the encryption example we gave above, the set of reachable relevant states is just the set of valid keys, which should be easy to specify.

In earlier work, using a slightly different HA model, we showed that SAC and functional consistency ensure correctness only when the HA is *strongly connected (SC)*, that is, when there exists a sequence of state transitions from every reachable state to every other reachable state. The same is true here.

Lemma 3 (Completeness of SAC + FC + SC)**.** *If an HA is strongly connected and single-action correct and has a bug, then it is not functionally consistent.*

However, strong functional consistency leads to an even stronger result.

Lemma 4 (Completeness of SAC + Strong FC)**.** *If an HA is single-action correct and has a bug, then it is not strongly functionally consistent.*

Finally, to address timeliness of results in addition to correctness, we define a notion of *responsiveness* for our model.

Definition 9 (Responsiveness)**.** *An HA is* responsive with respect to bound n *if, for all concrete initial states $s_0 \in S_{CI}$, sequences $\boldsymbol{in}$ of input batches, and input batches in, if*

- $\boldsymbol{s} = \mathit{StateSeq}(\boldsymbol{in}, s_0) = \langle s_0, \ldots, s_m \rangle$ *and*
- $\boldsymbol{s}' = \mathit{StateSeq}(\boldsymbol{in} \cdot in, s_0) = \langle s_0, \ldots, s_{m+l} \rangle$,

then $l \leq n$.

A. Decomposition for FC Checking

We now show how FC of a decomposed design can be derived from FC of its parts. We first give conditions under which two HAs can be composed.

Definition 10 (Functionally Composable). *Acc_1 and Acc_2 are* functionally composable *if: (i) $b_1 = b_2$; (ii) $O_1 = A_2 \times D_2$; (iii) $S_{C,1} \cap S_{C,2} = \emptyset$; (iv) $S_{R,1} = S_{R,2}$; and (v) $S_{N,1} = S_{Out,2} \times S'_N$ and $S_{N,2} = S_{In,1} \times S'_N$ for some S'_N.*

Note in particular that composability requires that the outputs of Acc_1 match the inputs of Acc_2. We also require that the two HAs have isomorphic memory states, which is ensured by including $S_{Out,2}$ in the non-relevant states of Acc_1 and $S_{In,1}$ in the non-relevant states of Acc_2. In order to map a memory state of Acc_1 to the corresponding memory state in Acc_2, we define a mapping function $\alpha : S_{M,1} \rightarrow S_{M,2}$ as follows: $\alpha(s_m) = (out(s_m), nrel(s_m)(1), rel(s_m), (inp(s_m), nrel(s_m)(2)))$. We next define functional composition.

Definition 11 (Functional Composition, Sub-Accelerators). *Given functionally composable HAs Acc_1 and Acc_2, we define the* functional composition *$Acc_0 = Acc_2 \circ Acc_1$ (Acc_1 and Acc_2 are called* sub-accelerators *of Acc_0) as follows: $b_0 = b_1$, $A_0 = A_1$, $D_0 = D_1$, $O_0 = O_2$, $S_{C,0} = S_{C,1} \cup S_{C,2}$, $S_{M,0} = S_{M,1}$, $s_{c,I,0} = s_{c,I,1}$, $s_{c,F,0} = s_{c,F,2}$, $S_{m,I,0} = S_{m,I,1}$. The transition function is defined as follows. $T_0(s_c, s_m) =$*

- (i) *if $s_c \in S_{C,1}$ and $s_c \neq s_{c,F,1}$ then $T_1(s_c, s_m)$;*
- (ii) *if $s_c \in S_{C,2}$ then $T_2(s_c, \alpha(s_m))$; and*
- (iii) *if $s_c = s_{c,F,1}$ then $(s_{c,I,2}, \alpha(s_m))$.*

Definition 11 essentially states that an execution of $Acc_0 = Acc_2 \circ Acc_1$ is obtained by first running Acc_1 to completion, then passing the outputs of Acc_1 to the inputs of Acc_2, and then running Acc_2 to completion. As a variant of Definition 11, it is also possible to define functional composition where the sub-accelerators operate in parallel. This way, the sub-accelerators process non-overlapping parts of a given input batch and produce the respective non-overlapping parts of the output batch.

We now introduce a compositional version of FC.

Definition 12 (Strong FC for Decomposition (*FCD*)). *An HA Acc is* strongly functionally consistent for decomposition (strongly FCD) *if it is strongly functionally consistent and, in addition to $o(j) = o'(j')$, the property $rel(s_F) = rel(s'_F)$ holds in the conclusion of the implication in Definition 7.*

Note that strong FCD is stronger than strong FC. In order to stitch together results on sub-accelerators, we need to establish that not only the output but also the relevant memory state is the same after processing identical inputs. The following is clear from the definition.

Corollary 1. *If an HA Acc is strongly FCD, then Acc is strongly FC.*

We now show that composition preserves strong FCD and then state our main result.

Lemma 5 (Functional Composition and Strong FCD). *Let $Acc_0 = Acc_2 \circ Acc_1$. If both Acc_1 and Acc_2 are strongly FCD then Acc_0 is strongly FCD.*

Theorem 1 (Completeness of A-QED^2). *Let Acc_0, Acc_1, and Acc_2 be HAs such that $Acc_0 = Acc_2 \circ Acc_1$ and Acc_0 is single-action correct. If Acc_1 and Acc_2 are strongly FCD then Acc_0 is functionally correct.*

Theorem 1 states that A-QED^2 is complete. That is, by contraposition, if an HA Acc_0 has a bug, i.e., it is not functionally correct, then either Acc_1 or Acc_2 is not strongly FCD, and thus the bug can be detected by A-QED^2.

Note that there is no corresponding soundness result. This is because it is possible to decompose a functionally consistent HA into functionally inconsistent sub-accelerators. However, as shown in Section V, this appears to be rare in practice, and here again we reiterate our position on design for verification and advocate that also sub-accelerators should be designed with functional consistency in mind.

Functional composition can easily be generalized to more than two sub-accelerators. Moreover, it can be applied recursively to further decompose sub-accelerators. If functional decomposition based on Definition 11 is not applicable to further decompose a sub-accelerator, then such a sub-accelerator can be decomposed using existing formal decomposition approaches, though these require significant manual effort. Our approach identifies conditions under which simple, automatable decomposition of FC checking is possible.

IV. A-QED^2 Functional Decomposition in Practice

We now present our implementation of A-QED^2, which builds on the theoretical framework of the previous section. We combine functional decomposition with checks for FC (dFC), SAC (dSAC), and responsiveness (dRB).

A. Decomposition for FC: dFC

dFC takes as input a non-interfering LCA design Acc (satisfying Definitions 1 and 2) together with designer-provided annotations (explained in this section). dFC decomposes Acc into sub-accelerators (following Definition 11). FC checks are run on the sub-accelerators and any counterexamples are reported. Note that the way in which Acc is actually decomposed into sub-accelerators has no influence on the completeness of A-QED^2 (Theorem 1). That said, FC checks may scale better for certain decompositions. While failing FC checks expose consistency issues at the sub-accelerator level, it is possible that they do not cause incorrect behaviors at the full Acc level. However, we did not observe any instances of this in our experiments.

Our dFC implementation relies on identifying *batch operations* in a given Acc. A batch operation operates on a vector of inputs, applying some action to each input in order to produce a vector of outputs. The input to a batch operation could be an intermediate output batch of another sub-accelerator or an input batch to Acc itself. A batch operation produces either an intermediate output batch which is subsequently processed by another sub-accelerator or an output batch of Acc itself.

We assume that Acc is expressed in a high-level language, specifically as a C/C++ program[1] that implements sequential computation of Acc outputs from Acc inputs.[2] Batch operations in the C/C++ program are identified by finding contiguous C/C++ statements called *functional blocks* that implement those batch operations. Each functional block represents a sub-accelerator.

We have developed a set of annotations by which the designer can help identify these functional blocks. Examples of such annotations are given in Listing 2 (extends Listing 1). It has two functional blocks corresponding to batch operations: lines 15-17 and 32-33.

Annotations are defined by particular keywords that are prefixed by "%" (and denoted in blue) in Listing 2. These annotations describe the compute and memory access patterns of the functional block as it transforms an input batch into an output batch. In practice, hardware designers already use similar annotations frequently, e.g., to express parallelization opportunities for HLS to generate efficient hardware. As a result, we expect manageable effort in creating such annotations to support dFC. The HLS research community is actively developing new techniques to automatically explore the HA design space and derive optimal design points together with appropriate parallelization and pipelining [54]–[56]. With tight integration of A-QED^2 with HLS, we expect that it will be possible to generate dFC annotations with low effort.

Listing 2: C/C++ Annotation Example (AES Encryption)

```
1  #define BS ((1) << 12) // BUF_SIZE
2  #define UF 2 // UNROLL_FACTOR
3  #define US BS/UF // UNROLL_SIZE
4
5  void fun(int data[BS], int buf[UF][US], int key[2]){
6    int j, k;
7
8    %IN_SIZE 16 // variables per input batch element
9    %IN_BATCH_SIZE BS/IN_SIZE // input batch size
10   %BATCH_MEM_IN data // input batch source
11   %IN_ALLOC_RULE in(x) addr range =
12     [i*BS +  x*IN_SIZE :
13     i*BS + (x + 1)*IN_SIZE]  // BATCH_MEM_IN layout
14   // ===ACC1 START===
15   for(j=0; j<UF; j++)
16     for(k = 0; k < BS/UF; k ++)
17       buf[j][k] = *(data + i*BS + j*US + k)^key[0];
18   // ===ACC1 END===
19   %OUT_SIZE 16 // variables per output batch element
20   %OUT_BATCH_SIZE BS/OUT_SIZE // output batch size
21   %BATCH_MEM_OUT buf // output batch source
22   %IN_ALLOC_RULE out(x) addr range =
23     [x/US][(x%US)*OUT_SIZE :
24     ((x + 1)%US)*OUT_SIZE] // BATCH_MEM_OUT layout
25
26   %IN_SIZE 16
27   %IN_BATCH_SIZE BS/IN_SIZE
28   %BATCH_MEM_IN buf
29   %IN_ALLOC_RULE in(x) addr range =
30     [(x%US)*IN_SIZE : ((x+1)%US)*IN_SIZE][x/US]
31   // ===ACC2 START===
32     for(j=0; j<UF; j++){
33       aes256_encrypt(local_key[j], buf[j]);}
34   // ===ACC2 END===
35   %OUT_SIZE 16
36   %OUT_BATCH_SIZE BS/OUT_SIZE
37   %BATCH_MEM_OUT buf
38   %OUT_ALLOC_RULE out(x) addr range =
39     [(x%US)*OUT_SIZE : ((x+1)%US)*OUT_SIZE][x/US]
40 }
```

From the annotations, we create sub-accelerators. For example, the annotations in Listing 2 generate two sub-accelerators: Acc_1 corresponding to the functional block in Lines 15-17 with annotations in Lines 8-13 and 19-24, and Acc_2 corresponding to the functional block in Lines 32-33 with annotations in Lines 26-30 and 35-39. For each sub-accelerator, we create an *A-QED*^{2} *module* for FC checking.[3] It generates symbolic inputs for the sub-accelerator and symbolically executes the corresponding functional block in order to produce symbolic expressions for the outputs. For strong FC checks (Definitions 6 and 7), the relevant states (Definition 1) must additionally be identified and explicitly constrained to be consistent across sub-accelerator calls processing two input batches. Identifying the relevant states is not necessary for intra-batch FC checks (discussed in the context of Lemma 2). For example, in sub-accelerator Acc_1 in Listing 2, *key[0]* is a relevant state element (distinct from the batch input *data*). Between two calls of Acc_1 during a strong FC check, *key[0]* must be consistent. In our implementation, we ignore reachability and allow all checks to start from fully symbolic initial states. This does not lead to spurious counterexamples in our experiments.

B. Decomposition for RB: dRB

The sub-accelerators for A-QED^2's RB checks (Definition 9) can be (and often are) different from those for FC because RB involves a much simpler check: *some* output is produced within the response bound n. We expect n to be provided by the designer for the top-level accelerator. We then use the same bound n for each sub-accelerator. The rationale is that if a sub-accelerator fails an RB check, then the full accelerator would also fail the same RB check.

For dRB, we generate a static single assignment (SSA) representation of the design. We then apply a *sliding window algorithm* to dynamically generate sub-accelerators. Lines of code in the SSA that fall within a certain *window* W form the sub-accelerator. Due to SSA form, the inputs of this sub-accelerator are variables that are never updated or assigned in W while the outputs are the variables which update variables outside W. The current size of W is given by the number of LOCs that fit in W, and it changes dynamically during a run of the algorithm to incorporate the largest sub-accelerator that will fit the BMC tool. Once the sub-accelerator is verified, W slides by δ LOCs (δ is a parameter) and adjusts its boundary to get the next largest sub-accelerator that can be verified. We synthesize that sub-accelerator using HLS (since some responsiveness bugs only manifest after HLS) and then run RB checks using BMC. The initial states of each generated

[1] HAs expressed in Verilog or SystemC can be converted into C/C++, and then our dFC implementation can be applied. We do this in Sec. V.

[2] Existing HLS tools (e.g., Xilinx Vivado HLS, Mentor Catapult HLS) can then optimize Acc, incorporate appropriate pipelining and parallelism, and produce Verilog for subsequent logic synthesis and physical design steps. Such HLS-based HA design flows are becoming increasingly common.

[3] See the online appendix [53] for details.

sub-accelerator are left unconstrained (i.e., fully symbolic) in order to analyze all possible behaviors. The specific size of W and its position in the SSA code change dynamically as dRB proceeds. dRB terminates when W reaches the end of the SSA code or if at any time an RB check fails.

C. Decomposition for SAC: dSAC

As mentioned above, and as will be shown in the next section, many bugs can be detected using only dFC and dRB. The advantage of this is that both of these checks can be run without any functional specification. dSAC completes the story, but at the cost of requiring specifications. We use standard functional decomposition techniques (essentially, writing preconditions, invariants, and postconditions) to decompose SAC checks. One feature of dSAC is that only a single input in a batch needs be checked—all other inputs in the batch can be set to constants (we use zero in our experiments). This makes both writing the properties and checking them much simpler. The non-input part of the initial state for each check is again kept fully symbolic for simplicity. If a sub-accelerator is too big, we further decompose it using finer-grained functional blocks.

V. Experimental Results

We demonstrate the practicality and effectiveness of A-QED^2 for 109 (buggy) versions of several non-interfering LCAs,[4] including open-source industrial designs [12]. We selected these designs for the following reasons:

- They cover a wide variety of HAs (neural nets, image processing, natural language processing, security). Most are too large for existing off-the-shelf formal tools.
- They have been thoroughly verified (painstakingly) using state-of-the-art simulation-based verification techniques. Thus, we can quantify the thoroughness of A-QED^2.
- With access to buggy versions, we did not have to artificially inject bugs. Bugs we encountered include incorrect initialization, incorrect memory accesses, incorrect array indexing, and unresponsiveness in HLS-generated designs.

Many of the designs were already available in sequential C or C++. We converted Verilog and SystemC designs into sequential C. To facilitate dFC, we manually inserted annotations (like those in Listing 2). For A-QED FC, we used CBMC for all designs originally represented in sequential C or C++. For designs in Verilog and SystemC, we used Cadence JasperGold (SystemC designs converted to Verilog via HLS). For A-QED^2 FC and SAC checks, we used CBMC version 5.10 [66]. For A-QED and A-QED^2 RB checks, we used Cadence JasperGold version 2016.09p002 on Verilog designs generated by the HLS tools used by the designers. Lastly, we used Frama-C [67] to check for initialization and out-of-bounds bugs on the entire C/C++ designs. We ran all our experiments on Intel Xeon E5-2640 v3 with 128GBytes of DRAM.

Tables I, II, and III summarize our results. We present comparisons between A-QED^2 (dFC, dRB, dSAC) and A-QED (FC, RB, SAC). Table I also compares A-QED^2 intra-batch FC vs. A-QED^2 strong FC (cf. details in the online appendix [53]).

Observation 1: HAs from various domains (including industry) show that non-interfering LCAs are highly common.

Observation 2: The vast majority of the studied HAs are too big for existing off-the-shelf formal verification tools, for both A-QED and conventional formal property verification.

Observation 3: Table I shows that A-QED^2 intra-batch FC checks detected bugs inside sub-accelerators (with batch sizes > 1) very quickly—under a minute for almost all of the designs, and just over a minute for nv_large. For most batch-mode sub-accelerators—except two for each of the following four designs (amounting to eight sub-accelerators in total): grayscale64, grayscale32, mean128, and mean32—intra-batch dFC checks were easily completed using off-the-shelf formal tools. Strong FC checks incur more complexity. Hence, the formal tool timed out after 12 hours for 62 sub-accelerators when running strong FC checks, distributed across multiple designs. Empirically, we found that intra-batch FC checks detected all bugs that were detected by strong FC checks.

Observation 4: A-QED^2 RB and A-QED^2 SAC are also highly effective in detecting bugs inside sub-accelerators. For the first 11 designs (**AES** to **gsm**) in Table II, we do not expect unresponsiveness bugs (confirmed by simulations). Hence, A-QED^2 RB checks ran for 12 hours (for increasingly longer input sequences) without detecting unresponsiveness. For designs with RB bugs, A-QED^2 RB checks on sub-accelerators were able to detect those in less than 11 minutes on average. For A-QED^2 dSAC, we observed that a significant fraction (26 out of 46 bugs (56%)) of these bugs were also detected by A-QED^2 FC checks. Thus, FC alone is effective at catching a wide variety of bugs.

Observation 5: A-QED^2 detected all bugs that were detected by conventional (simulation-based) verification techniques. Further, all counterexamples produced from verifying sub-accelerators corresponded to real accelerator-level bugs. Compared with traditional simulation-based verification, we report a $\sim 5X$ improvement in verification effort on the average, with a $\sim 9X$ improvement for the large, industrial NVDLA designs. The overhead of inserting our annotations for dFC can be small compared to what designers already insert to optimize the design. For ISmartDNN, for example, the total number of annotations is 304, which is 2.8% of the total lines of code of the design. In the code of the HLS designs we considered, pragmas amount to 11% on average. We also observe a $\sim 60X$ improvement in average verification runtime compared to conventional simulations.[5]

VI. Conclusion

Our theoretical and experimental results demonstrate that A-QED^2 is an effective and practical approach for verification

[4] See the online appendix [53] for design details and the software artifact [65].

[5] The conventional verification effort for NVDLA was based on start and end commit dates in its nv_small Github repository. The conventional verification runtime for NVDLA, ISmartDNN, and dnn HAs were obtained by running the available simulation tests on our platform. The remaining runtime and effort information were provided by the designers.

Design (#Gates) (#Versions)	A-QED FC	A-QED^2 dFC: Intra-batch FC			A-QED^2 dFC: Strong FC		
94 versions in table, 15 in caption†	**Avg. RT (min)**	**Avg. RT (min)**	**#Bugs**	**#Sub-Acc.(T/P/C/B)**	**Avg. Runtime (min)**	**#Bugs**	**#Sub-Acc.(T/P/C/B)**
AES [50] (382k) (4)	OOM	0.97	4	8 / 7 / 7 / 4	timeout	0	8 / 7 / 2 / 0
ISmartDNN [57] (42M) (3)	timeout	0.10	2	38 / 5 / 5 / 2	0.18	2	38 / 5 / 2 / 2
grayscale128 [33] (351k) (5)	timeout	0.03	3	3 / 3 / 2 / 2	0.07	3	3 / 3 / 2 / 2
grayscale64 [33] (194k) (5)	timeout	0.02	3	3 / 3 / 2 / 2	0.02	3	3 / 3 / 2 / 2
grayscale32 [33] (106k) (5)	8.20	<0.01	5	3 / 3 / 3 / 3	0.30	5	3 / 3 / 3 / 3
mean128 [33] (202k) (5)	timeout	0.35	3	3 / 3 / 2 / 2	0.17	3	3 / 3 / 2 / 2
mean64 [33] (104k) (5)	timeout	0.38	3	3 / 3 / 2 / 2	0.13	3	3 / 3 / 2 / 2
mean32 [33] (54k) (5)	5.53	0.17	5	3 / 3 / 3 / 3	0.33	5	3 / 3 / 3 / 3
dnn [58] (2M) (11)	timeout	0.03	5	34 / 14 / 14 / 5	0.13	5	34 / 14 / 8 / 5
nv_large [12] (16M) (23)	timeout	1.17	11	89 / 46 / 46 / 11	2.93	9	89 / 46 / 21 / 9
nv_small [12] (1M) (23)	timeout	0.07	11	89 / 46 / 46 / 11	1.03	11	89 / 46 / 26 / 11

TABLE I: **Avg**. **R**un**T**imes of FC checks for A-QED and A-QED^2. For A-QED^2, sub-accelerator counts are provided, including the **T**otal count that resulted from dFC decomposition, the count with batch sizes greater than one (i.e., **P**arallel), the count (with batch sizes greater than one) for which FC checks were successful on 1 and 2 batches for intra-batch FC and strong FC respectively, and the count for which **B**ugs were detected by FC checks. For A-QED FC, experiments could not complete FC check for a single batch in 12 hours (**timeout**) or exhibited out-of-memory (**OOM**) errors before timeout. **Average** runtimes result from dividing the time to detect all bugs by the number of bugs. †**keypair** [59], **gsm** [60], **HLSCNN** [61], **FlexNLP** [62], **Dataflow** [63], and **Opticalflow** [64] all time out for A-QED FC and do not contain any sub-accelerators with batch size greater than one. One OOB bug was detected in **gsm** and one initialization bug in **keypair**.

Design (#Gates) (#Versions) Total Versions = 109	A-QED RB	A-QED^2 dRB		
	Avg. RT (min)	**Avg. RT (min)**	**#Bugs**	**#Sub-Acc. (T/C/B)**
AES [50] (382k) (4)	timeout	No RB bug detected up to input sequence length between 11 and 24 depending on the design		13 / 13 / 0
ISmartDNN [57] (42M) (3)	timeout			32 / 32 / 0
grayscale128 [33] (351k) (5)	timeout			5 / 5 / 0
grayscale64 [33] (194k) (5)	timeout			5 / 5 / 0
grayscale32 [33] (106k) (5)				3 / 3 / 0
mean128 [33] (202k) (5)	timeout			5 / 5 / 0
mean64 [33] (104k) (5)	timeout			3 / 3 / 0
mean32 [33] (54k) (5)				1 / 1 / 0
dnn [58] (2M) (11)	timeout			5 / 5 / 0
keypair [59] (>200M) (1)	timeout			21 / 21 / 0
gsm [60] (8.8k) (1)	timeout			7 / 7 / 0
nv_large [12] (16M) (23)	timeout	No RB bugs expected		
nv_small [12] (1M) (23)	timeout			
HLSCNN [61] (323k) (2)	timeout	2.33	1	25 / 25 / 1
FlexNLP [62] (567k) (9)	timeout	10.77	9	15 / 15 / 9
Dataflow [63] (296k) (1)	0.45	0.25	1	9 / 9 / 1
Opticalflow [64] (555k) (1)	timeout	0.17	1	3 / 3 / 1

TABLE II: RB checks for A-QED and A-QED^2. For A-QED^2, sub-accelerator counts produced by dFC are provided, as in Table I. A-QED^2 RB checks are performed on all sub-accelerators regardless of batch size, so **P** is omitted compared to Table I. For A-QED RB, RB checks did not complete even for a input sequence length of 1 within 12 hours (**timeout**). Sub-accelerators for which RB checks for at least input sequence length of 1 was completed were considered **C**omplete. For the first 11 designs, from **AES** to **gsm**, no bugs related to unresponsiveness were detected by traditional simulation-based verification. Results are omitted for **nv_large** and **nv_small**; responsiveness related bugs generally result from parallelism and pipelining, both of which were lost in our manual translation of NVDLA from Verilog to sequential C code.

of large non-interfering LCAs. A-QED^2 exploits A-QED principles to decompose a given HA design into sub-accelerators such that A-QED can be naturally applied to the sub-accelerators. A-QED^2 is especially attractive for HLS-based HA design flows. A-QED^2 creates several promising research directions:

- Extension of our A-QED^2 experiments to include interfering LCAs (already covered by our theoretical results).
- Automation of dFC annotations via HLS techniques.
- dFC approaches beyond our current implementation.
- Further A-QED^2 scalability using abstraction.
- Extension of A-QED^2 beyond sequential (C/C++) code to include concurrent programs.
- Effectiveness of A-QED^2 for RTL designs (without converting them to sequential C/C++).
- Applicability of A-QED^2 beyond functional bugs (e.g., to detect security vulnerabilities in HAs).
- Comparison of A-QED^2 and conventional decomposition.
- Identifying conditions under which A-QED^2 is sound.

Design (#Gates) (#Versions) Total Versions = 109	A-QED^2 dSAC			
	Avg. RT (min)	**#Bugs**	**Bug overlap with dFC**	**#Sub-Acc. (T/C/B)**
AES [50] (382k) (4)	0.12	0	0	8 / 8 / 0
ISmartDNN [57] (42M) (3)	0.22	3	2	38 / 38 / 3
grayscale128 [33] (351k) (5)	0.04	2	2	3 / 2 / 2
grayscale64 [33] (194k) (5)	0.01	2	2	3 / 2 / 2
grayscale32 [33] (106k) (5)	<0.01	2	2	3 / 3 / 2
mean128 [33] (202k) (5)	0.21	2	2	3 / 2 / 2
mean64 [33] (104k) (5)	<0.01	2	2	3 / 2 / 2
mean32 [33] (54k) (5)	<0.01	2	2	3 / 3 / 2
dnn [58] (2M) (11)	0.01	6	0	34 / 14 / 6
keypair [59] (>200M) (1)	timeout	0	0	14 / 14 / 0
gsm [60] (8.8k) (1)	timeout	0	0	5 / 5 / 0
nv_large [12] (16M) (23)	0.84	12	6	89 / 89 / 12
nv_small [12] (1M) (23)	0.11	12	6	89 / 50 / 12
HLSCNN [61] (323k) (2)	0.45	1	0	25 / 11 / 1
FlexNLP [62] (567k) (9)	timeout	0	0	21 / 21 / 0
Dataflow [63] (296k) (1)	timeout	0	0	8 / 8 / 0
Opticalflow [64] (555k) (1)	timeout	0	0	14 / 14 / 0

TABLE III: SAC checks for A-QED^2. Sub-accelerator counts produced by dSAC are provided, as in Table I. A-QED^2 SAC checks were performed on all sub-accelerators regardless of batch size, so **P** is omitted compared to Table I.

Acknowledgment

This work was supported by the DARPA POSH program (grant FA8650-18-2-7854), NSF (grant A#:1764000), and the Stanford SystemX Alliance. We thank Prof. David Brooks, Thierry Tambe and Prof. Gu-Yeon Wei from Harvard University, and Kartik Prabhu and Prof. Priyanka Raina from Stanford University for their design contributions in our experiments.

REFERENCES

[1] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, “Accelerator-rich architectures: Opportunities and progresses,” in *Proc. DAC*. IEEE, 2014, pp. 1–6.

[2] L. P. Carloni, “The Case for Embedded Scalable Platforms,” in *Proc. DAC*. IEEE, 2016, pp. 1–6.

[3] W. J. Dally, Y. Turakhia, and S. Han, “Domain-Specific Hardware Accelerators,” *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, 2020.

[4] M. Hill and V. J. Reddi, “Accelerator-level Parallelism,” *CoRR*, vol. abs/1907.02064, 2019, https://arxiv.org/abs/1907.02064.

[5] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, “The Design Process for Google’s Training Chips: TPUv2 and TPUv3,” *IEEE Micro*, 2021.

[6] H. D. Foster, “Trends in functional verification: a 2014 industry study,” in *Proc. DAC*. ACM, 2015, pp. 48:1–48:6.

[7] B. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, “Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification,” *ACM Trans. Design Autom. Electr. Syst.*, vol. 24, no. 1, pp. 10:1–10:24, 2019.

[8] E. Singh, F. Lonsing, S. Chattopadhyay, M. Strange, P. Wei, X. Zhang, Y. Zhou, D. Chen, J. Cong, P. Raina, Z. Zhang, C. W. Barrett, and S. Mitra, “A-QED Verification of Hardware Accelerators,” in *Proc. DAC*. IEEE, 2020, pp. 1–6.

[9] E. G. Cota, P. Mantovani, G. D. Guglielmo, and L. P. Carloni, “An Analysis of Accelerator Coupling in Heterogeneous Architectures,” in *Proc. DAC*. ACM, 2015, pp. 202:1–202:6.

[10] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, “Architecture support for accelerator-rich CMPs,” in *Proc. DAC*. ACM, 2012, pp. 843–849.

[11] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, “Bounded model checking using satisfiability solving,” *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.

[12] NVIDIA, “NVIDIA Deep Learning Accelerator,” http://nvdla.org/primer.html, 2021, [Online]. Accessed: August 2021.

[13] K. A. Campbell, D. Lin, L. He, L. Yang, S. T. Gurumani, K. Rupnow, S. Mitra, and D. Chen, “Hybrid Quick Error Detection: Validation and Debug of SoCs Through High-Level Synthesis,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 7, pp. 1345–1358, 2019.

[14] Y. Chi, Y. Choi, J. Cong, and J. Wang, “Rapid Cycle-Accurate Simulator for High-Level Synthesis,” in *Proc. FPGA*. ACM, 2019, pp. 178–183.

[15] IEEE, “IEEE Standard for Universal Verification Methodology Language Reference Manual,” *IEEE Std 1800.2-2017*, pp. 1–472, 2017.

[16] Y. Choi, Y. Chi, J. Wang, and J. Cong, “FLASH: Fast, Parallel, and Accurate Simulator for HLS,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4828–4841, 2020.

[17] S. Dai, A. Klinefelter, H. Ren, R. Venkatesan, B. Keller, N. R. Pinckney, and B. Khailany, “Verifying High-Level Latency-Insensitive Designs with Formal Model Checking,” *CoRR*, vol. abs/2102.06326, 2021. [Online]. Available: https://arxiv.org/abs/2102.06326

[18] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[19] D. Giannakopoulou, K. S. Namjoshi, and C. S. Pasareanu, “Compositional Reasoning,” in *Handbook of Model Checking*. Springer, 2018, pp. 345–383.

[20] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh, “Assume-Guarantee Verification of Source Code with Design-Level Assumptions,” in *Proc. ICSE*. IEEE Computer Society, 2004, pp. 211–220.

[21] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu, “Learning Assumptions for Compositional Verification,” in *Proc. TACAS*, ser. LNCS, vol. 2619. Springer, 2003, pp. 331–346.

[22] A. Gupta, K. L. McMillan, and Z. Fu, “Automated assumption generation for compositional verification,” *Formal Methods in System Design*, vol. 32, no. 3, pp. 285–301, 2008.

[23] R. Jhala and K. L. McMillan, “Microarchitecture Verification by Compositional Model Checking,” in *Proc. CAV*, ser. LNCS, vol. 2102. Springer, 2001, pp. 396–410.

[24] C. Y. Cho, V. D’Silva, and D. Song, “BLITZ: Compositional bounded model checking for real-world programs,” in *Proc. ASE*. IEEE, 2013, pp. 136–146.

[25] H. Koo and P. Mishra, “Functional test generation using design and property decomposition techniques,” *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 4, pp. 32:1–32:33, 2009.

[26] R. B. Jones, C. H. Seger, and D. L. Dill, “Self-Consistency Checking,” in *Proc. FMCAD*, ser. LNCS, vol. 1166. Springer, 1996, pp. 159–171.

[27] S. Katz, O. Grumberg, and D. Geist, “"Have I written enough Properties?" - A Method of Comparison between Specification and Implementation,” in *Proc. CHARME*, ser. LNCS, vol. 1703. Springer, 1999, pp. 280–297.

[28] K. Claessen, “A Coverage Analysis for Safety Property Lists,” in *Proc. FMCAD*. IEEE, 2007, pp. 139–145.

[29] H. Chockler, O. Kupferman, and M. Y. Vardi, “Coverage Metrics for Temporal Logic Model Checking,” in *Proc. TACAS*, ser. LNCS, vol. 2031. Springer, 2001, pp. 528–542.

[30] D. Große, U. Kühne, and R. Drechsler, “Estimating functional coverage in bounded model checking,” in *Proc. DATE*. EDA Consortium, San Jose, CA, USA, 2007, pp. 1176–1181.

[31] H. Chockler, D. Kroening, and M. Purandare, “Coverage in interpolation-based model checking,” in *Proc. DAC*. ACM, 2010, pp. 182–187.

[32] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, “Kami: a platform for high-level parametric hardware specification and its modular verification,” *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 24:1–24:30, 2017.

[33] L. Piccolboni, G. Di Guglielmo, and L. P. Carloni, “KAIROS: Incremental Verification in High-Level Synthesis through Latency-Insensitive Design,” in *Proc. FMCAD*. IEEE, 2019, pp. 105–109.

[34] U. Kühne, S. Beyer, J. Bormann, and J. Barstow, “Automated formal verification of processors based on architectural models,” in *Proc. FMCAD*. IEEE, 2010, pp. 129–136.

[35] M. Soeken, U. Kühne, M. Freibothe, G. Fey, and R. Drechsler, “Automatic property generation for the formal verification of bus bridges,” in *Proc. DDECS*. IEEE, 2011, pp. 417–422.

[36] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke, “Advanced verification by automatic property generation,” *IET Comput. Digit. Tech.*, vol. 3, no. 4, pp. 338–353, 2009.

[37] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, “Symbolic Model Checking without BDDs,” in *Proc. TACAS*, ser. LNCS, vol. 1579. Springer, 1999, pp. 193–207.

[38] D. Lin, E. Singh, C. Barrett, and S. Mitra, “A structured approach to post-silicon validation and debug using symbolic quick error detection,” in *Proc. ITC*. IEEE, 2015, pp. 1–10.

[39] E. Singh, D. Lin, C. Barrett, and S. Mitra, “Logic Bug Detection and Localization Using Symbolic Quick Error Detection,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2018.

[40] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. R. Fadiheh, D. Stoffel, W. Kunz, C. W. Barrett, W. Ecker, and S. Mitra, “Symbolic QED Pre-Silicon Verification for Automotive Microcontroller Cores: Industrial Case Study,” in *Proc. DATE*. IEEE, 2019, pp. 1000–1005.

[41] F. Lonsing, K. Ganesan, M. Mann, S. S. Nuthakki, E. Singh, M. Srouji, Y. Yang, S. Mitra, and C. W. Barrett, “Unlocking the Power of Formal Hardware Verification with CoSA and Symbolic QED: Invited Paper,” in *Proc ICCAD*. ACM, 2019, pp. 1–8.

[42] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz, “Symbolic quick error detection using symbolic initial state for pre-silicon verification,” in *Proc. DATE*. IEEE, 2018, pp. 55–60.

[43] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. W. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz, “Gap-free Processor Verification by S^2QED and Property Generation,” in *Proc. DATE*. IEEE, 2020, pp. 526–531.

[44] M. R. Fadiheh, D. Stoffel, C. W. Barrett, S. Mitra, and W. Kunz, “Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking,” in *Proc. DATE*. IEEE, 2019, pp. 994–999.

[45] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, “A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors,” in *Proc. DAC*. IEEE, 2020, pp. 1–6.

[46] G. Barthe, P. R. D’Argenio, and T. Rezk, “Secure Information Flow by Self-Composition,” in *Proc. CSFW-17*. IEEE, 2004, pp. 100–114.

[47] G. Barthe, J. M. Crespo, and C. Kunz, “Relational Verification Using Product Programs,” in *Proc. FM*, ser. LNCS, vol. 6664. Springer, 2011, pp. 200–214.

[48] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, “Verifying Constant-Time Implementations,” in *Proc. USENIX*. USENIX Association, 2016, pp. 53–70.

[49] W. Yang, Y. Vizel, P. Subramanyan, A. Gupta, and S. Malik, “Lazy Self-composition for Security Verification,” in *Proc. CAV*, ser. LNCS, vol. 10982. Springer, 2018, pp. 136–156.

[50] J. Cong, P. Wei, C. H. Yu, and P. Zhou, “Bandwidth optimization through on-chip memory restructuring for HLS,” in *Proc. DAC*. IEEE, 2017, pp. 1–6.

[51] R. M. Keller, “Formal Verification of Parallel Programs,” *Commun. ACM*, vol. 19, no. 7, pp. 371–384, 1976.

[52] ——, “A Fundamental Theorem of Asynchronous Parallel Computation,” in *Parallel Processing, Proc. Sagamore Computer Conference*, ser. LNCS, vol. 24. Springer, 1974, pp. 102–112.

[53] S. Chattopadhyay, F. Lonsing, L. Piccolboni, D. Soni, P. Wei, X. Zhang, Y. Zhou, L. Carloni, D. Chen, J. Cong, R. Karri, Z. Zhang, C. Trippel, C. Barrett, and S. Mitra, “Scaling Up Hardware Accelerator Verification using A-QED with Functional Decomposition,” *CoRR*, vol. abs/2108.06081, 2021, FMCAD 2021 proceedings version with appendix. [Online]. Available: https://arxiv.org/abs/2108.06081

[54] S. Wang, Y. Liang, and W. Zhang, “FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs,” in *Proc. DAC*. ACM, 2017, pp. 27:1–27:6.

[55] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, “COMBA: A Comprehensive Model-Based Analysis Framework for High Level Synthesis of Real Applications,” in *Proc. ICCAD*. IEEE, 2017, pp. 430–437.

[56] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, “Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators,” in *Proc. DAC*. ACM, 2016, pp. 136:1–136:6.

[57] X. Zhang, H. Lu, C. Hao, J. Li, B. Cheng, Y. Li, K. Rupnow, J. Xiong, T. S. Huang, H. Shi, W. Hwu, and D. Chen, “SkyNet: a Hardware-Efficient Method for Object Detection and Tracking on Embedded Systems,” in *Proc. MLSys*. mlsys.org, 2020.

[58] M. Giordano, K. Prabhu, K. Koul, R. M. Radway, A. Gural, R. Doshi, Z. F. Khan, J. W. Kustin, T. Liu, G. B. Lopes, V. Turbiner, W.-S. Khwa, Y.-D. Chih, M.-F. Chang, G. Lallement, B. Murmann, S. Mitra, and P. Raina, “CHIMERA: A 0.92 TOPS, 2.2 TOPS/W Edge AI Accelerator with 2 MByte On-Chip Foundry Resistive RAM for Efficient Training and Inference,” in *Proc. VLSI*. IEEE, 2021, pp. 1–2.

[59] K. Basu, D. Soni, M. Nabeel, and R. Karri, “NIST Post-Quantum Cryptography- A Hardware Evaluation Study,” IACR Cryptology ePrint Archive, Report 2019/047, 2019, https://eprint.iacr.org/2019/047.

[60] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, “Chstone: A benchmark program suite for practical c-based high-level synthesis,” in *Proc. ISCAS*. IEEE, 2008, pp. 1192–1195.

[61] P. N. Whatmough, S. K. Lee, M. Donato, H. Hsueh, S. L. Xi, U. Gupta, L. Pentecost, G. G. Ko, D. M. Brooks, and G. Wei, “A 16nm 25mm^2 SoC with a 54.5x Flexibility-Efficiency Range from Dual-Core Arm Cortex-A53 to eFPGA and Cache-Coherent Accelerators,” in *Proc. VLSI*. IEEE, 2019, p. 34.

[62] T. Tambe, E. Yang, G. G. Ko, Y. Chai, C. Hooper, M. Donato, P. N. Whatmough, A. M. Rush, D. Brooks, and G. Wei, “A 25mm^2 SoC for IoT Devices with 18ms Noise-Robust Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET,” in *Proc. ISSCC*. IEEE, 2021, pp. 158–160.

[63] Y. Chi, Y. Choi, J. Cong, and J. Wang, “Rapid Cycle-Accurate Simulator for High-Level Synthesis,” in *Proc. FPGA*. ACM, 2019, pp. 178–183.

[64] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. K. Srivastava, H. Jin, J. Featherston, Y. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, “Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs,” in *Proc. FPGA*. ACM, 2018, pp. 269–278.

[65] “A-QED^2 Software Artifact,” 2021. [Online]. Available: https://github.com/upscale-project/aqed-decomp-FMCAD2021/

[66] D. Kroening and M. Tautschnig, “CBMC - C bounded model checker - (competition contribution),” in *Proc. TACAS*, ser. LNCS, vol. 8413. Springer, 2014, pp. 389–391.

[67] “Frama-C,” https://frama-c.com/, 2021, [Online]. Accessed: August 2021.