Automatic Hard Block Inference on FPGAs

Adrian Willenbücher
Embedded Systems Group
University of Kaiserslautern
willenbuecher@cs.uni-kl.de

Klaus Schneider
Embedded Systems Group
University of Kaiserslautern
klaus.schneider@cs.uni-kl.de

Abstract—Modern FPGAs often provide a number of highly optimized hard IP blocks with certain functionalities. However, manually instantiating these blocks is both time-consuming and error-prone, in particular, if only a part of the functionality of the IP block is used. To solve this problem, we developed an algorithm to automatically replace a selected combinational subset of a hardware design with a correct instantiation of a given IP block. Both the IP block and the part of the hardware circuit to be replaced are specified using arithmetic and Boolean operators. Our method is based on higher-order E-unification with an equational theory of arithmetic and Boolean laws. To demonstrate the effectiveness and efficiency of our approach, we present preliminary experiments with various circuits.

I. INTRODUCTION

One way to optimize the performance, size, and energy efficiency of FPGA designs is to make use of hard IP blocks – non-programmable blocks providing a fixed, but often flexible functionality – which are abundant on modern FPGAs. For example, such hard blocks can be digital signal processing (DSP) elements which typically contain at least a multiplier and an adder/subtractor.

Due to their flexibility, these hard blocks tend to be complicated to use: they come with extensive documentation and typically, only a small subset of inputs, outputs and options are needed for a specific instantiation of such a block. A simple one-line statement like

\[ p = s ? (a \cdot b - c) : (a \cdot b + c); \]

might require a verbose hard block instantiation, which takes up screen space, obfuscates the hardware designer’s actual intent, and potentially introduces bugs. Therefore, as an alternative to manual instantiation, FPGA synthesis software is capable of a very limited form of automatic inference, but even current commercial software like Xilinx Vivado is unable to recognize that the calculation above can be replaced by a DSP block without any additional logic.

A third alternative is to use IP core generators: for a given task, they create an IP core which contains hard blocks and glue logic for instantiation. However, also this method has its drawbacks:

- It obviously requires an IP core generator for the considered problem.
- The IP core still has to be instantiated manually, resulting in more verbose and less intuitive code.

A special case of this approach is software for implementing DSP applications (for example Altera’s DSP Builder), which takes a high-level system description and generates an optimized implementation for a given FPGA target. Although these tools are powerful, they are tailored to more complex problems and still require to manually incorporate the results into the design.

For this reason, more effective ways to automatically infer an instantiation of a hard block are needed. Usually, the hardware designer has a rather good intuition about which parts of the circuit can be replaced by certain hard blocks. Ideally, the matching of the hard block to the circuit should be done automatically without further interaction of the user. In practice, however, there can be multiple solutions offering compromises between, e.g., frequency and area. In this case, those solutions should be presented to the user, thereby enabling a form of design space exploration. Most of our approach will also work with soft IP cores, but for now, we focus on hard IP blocks.

In this paper, we present a novel approach to determine correct instantiations of given hard IP blocks to implement desired combinational circuits. To this end, we employ higher-order equational unification as a formal basis. We provide key information about our implementation, as well as preliminary experimental results. In Section II, previous work related to hard block instantiation is discussed. Our approach is described in Section III-A, an introduction to higher-order E-unification is given in Section III-B, and Section III-C contains information about our implementation. Finally, experimental results are discussed in Section IV, followed by a conclusion in Section V.

II. RELATED WORK

To the best of our knowledge, the problem of flexible, automatic hard block instantiation has not been treated before in its full generality. Although synthesis software by FPGA vendors can recognize, e.g., simple multiplication/addition combinations, it fails when being confronted with slightly more complex situations. There exist, however, some approaches which bear some resemblance to the problem treated in this paper.

A. Boolean Matching

Given two Boolean functions with the same number of inputs, the Boolean matching problem consists of computing a permutation of the inputs such that these functions become semantically equivalent [1]. Sometimes, permutation of the outputs (for functions with more than one output) or negation of inputs and/or outputs is allowed as well. Boolean matching is typically used for library binding, i.e., transforming a logic
representation to a netlist of cells from a given library [2], or for FPGA technology mapping [3]. Traditionally, Boolean matching was targeted at small functions with up to about 20 inputs [4], which is sufficient for standard cell binding [5]. Recently, however, methods to efficiently handle large problem instances have been found, so that even functions with hundreds of inputs and outputs can be matched within seconds [6].

Unfortunately, Boolean matching is not well suited for hard block instantiation, mainly for two reasons: First, hard blocks tend to have more input operands than the circuits they replace (for example, control inputs are set to a default or to constant values). A similar argument applies to outputs. Second, even if all inputs are used, they are often wider than the inputs of the circuit they replace, and thus have to be sign- or zero-extended. Also, since the inputs of the hard block are wider than needed, so are the outputs (in a typical case).

Therefore, the two functions representing the circuit and the hard block typically have a different number of input and output bits, and can thus not be matched this way. We did not find a way to modify the algorithms in [6] to handle functions with different numbers of Boolean inputs and outputs. For the special case where the hard block and the circuit have the same number of inputs and outputs, we tried to compare our method with Katebi’s and Markov’s [6] tool, which is integrated in ABC1 [7]. Unfortunately, we could not run our experiments due to a failing assertion in ABC’s backend.

B. Sketching

In sketching, an incomplete version of an optimized algorithm (the ‘sketch’) with ‘holes’ indicating not yet determined details, is automatically completed according to a reference implementation [8]. In its first realization, this concept was severely restricted: it could only handle bit-streaming programs which were built from filters that were connected by FIFO channels. An improved method for combinational programs was presented in [9]. However, the holes in this approach are still restricted to constants which makes it impossible to replace them with arbitrary expressions, and it cannot add logic which processes the program’s output.

C. Other Related Work

*Odin II* [10], a framework for Verilog synthesis, is able to map multiplications of arbitrary size to one or more hard block multipliers of a given size. However, it does not seem to be capable of mapping more complex operations to advanced hard blocks (like DSP elements).

Model checking of incomplete designs determines whether a design containing ‘black boxes’ (i.e., not yet available parts) can be still be completed to fulfill certain properties [11]. If this is the case, then there exists an implementation for these missing parts so that the design meets the specifications; otherwise, the other parts of the design are known to be incorrect. A similar problem is controller synthesis where one tries to find for a given system $K$ and its specification $\phi$ another system $C$ such that the combined system $K \parallel C$ satisfies $\phi$ [12].

III. Approach

The workflow for inferring a hard block using our algorithm is outlined by the following steps:

1) Identification of the part of the hardware design which should be completely or partially implemented by a hard block, along with selecting a suitable hard block. This is done manually at design time (i.e., before synthesis), but it could conceivably be done by a heuristic during synthesis (for example, by choosing multipliers and some surrounding logic).

2) A sequence of semantically correct instantiations is then automatically produced, until a solution of sufficient quality is found, e.g., until a time limit is reached, or until the developer stops the search. This algorithm is our contribution and will be discussed in detail.

3) Each solution is evaluated using fast estimations for a given metric (e.g., area). The best found is selected automatically if there’s just one criterion; if there are several criteria (e.g., delay and area), the developer has to choose one instantiation from the set of Pareto optimal solutions.

4) The original circuit is replaced by the hard block instantiation code before synthesis.

Although strictly speaking the dedicated carry logic in modern FPGAs qualifies as a hard block, it is not in our focus due to the low complexity of those circuits, and because there exist more suitable approaches (for example [3], [13]).

A. Preliminaries

Unless noted otherwise, we use the following conventions:

- $a, b, c, d, e$ denote constants and functions of any arity.
- $w, x, y, z$ denote first-order variables.
- $F, G, H$ denote higher-order variables.
- $u, v$ denote expressions.
- $\prod_{i} u_i$ is short hand for $u_1, \ldots, u_n$.
- $\lambda \prod_{i} u_i u$ denotes an anonymous function with parameters $x_1, \ldots, x_n$ and function body $u$.

An expression $e$ has the following syntax:

$$e ::= (e * e) | (e + e) | - e | (e = e) | (e < e) | (e ? e : e) | !e | F(e) | \text{variable} | \text{constant}$$

Unification literature typically uses terms instead of expressions, and treats operators as functions, e.g., writing $* (x, (y, z))$ instead of $x * (y + z)$. These two concepts are semantically interchangeable, and therefore only a matter of taste. We prefer to use expressions due to our algorithm’s application in hardware design.

A substitution is a mapping from variables to expressions; for example, $\sigma := \{ x \mapsto 4, y \mapsto (a + 2) \}$. When $\sigma$ is applied to an expression like $u := (y * z)$, the result is $\sigma(u) \equiv ((a + 2) * z)$. First-order variables are placeholders.

---

1http://www.eecs.berkeley.edu/~alanmi/abc
for values or expressions, while second-order variables are placeholders for functions. For example, given the expression $F(2, 3)$, then $F$ can be assigned any function which takes two arguments, e.g., for $\sigma \equiv \{ F \mapsto \lambda xy.(y * x + 5) \}$, we get $\sigma(F(2, 3)) \equiv (\lambda xy.(y * x + 5))(2, 3) \equiv (3 * 2 + 5)$. 

We model both, the hard block as well as the circuit to be replaced with the block, as symbolic expressions. The arithmetic operators do not overflow, which makes it easier to handle them symbolically; it is clear that not all hard blocks or circuits behave that way and we are working on a suitable way of supporting their semantics as well.

In the expression modeling the circuit, the inputs are represented by fresh, uninterpreted constant symbols. In contrast, the inputs of the hard block are represented by variables, which makes it possible to assign arbitrary expressions to the inputs of the block. Given the expressions $u$ for the circuit and $v$ for the hard block, we have to find a substitution $\sigma$ such that $u$ becomes semantically equivalent to $\sigma(v)$. This process is called matching. Matching is closely related to unification, where $\sigma$ has to satisfy $\sigma(u) \equiv \sigma(v)$. Note that in our case $\sigma(u) \equiv u$, since $u$ does not contain any variables. Therefore, we can use unification algorithms to find $\sigma$.

Although there exist algorithms for certain forms of semantic unification, these are limited to different subsets of semantic equivalence and are not sufficient for our goal. We therefore have to model equivalence of expressions using arithmetic and logic laws (e.g., commutativity and associativity of addition and multiplication), and employ equational unification (E-unification) to find a solution.

For example, the statement mentioned in Section I, which corresponds to the circuit shown in Fig.1, is modeled by the expression $s : (a * b + (c)) : (a * b + c)$ where $a, b, c$ and $s$ are ‘fresh’ constants. The simple DSP block shown in Fig.2 can be modeled as $x * y + (w ? - z : z)$. It can replace the subcircuit by using the input assignments $\{x \mapsto a, y \mapsto b, z \mapsto c, w \mapsto s\}$, resulting in the expression $a * b + (s ? - c : c)$ which is semantically equivalent to the above subcircuit. Now consider the expression $a * (b * c)$. Using the same hard block, the following substitution is a solution: $\{x \mapsto (a * b), y \mapsto c, z \mapsto 0, w \mapsto 0\}$, but also $\{x \mapsto 0, y \mapsto 0, z \mapsto (a * (b * c)), w \mapsto 0\}$ is a valid, but suboptimal solution.

In some cases, a circuit can be replaced by a hard block if logic is added not just to the block’s inputs, but also to its outputs; this concept is illustrated in Fig.3. For example, there exists no substitution which unifies the expression $(a * b) < c$ (see Fig.4a) with the above DSP block, because the comparison operator $<$ cannot be moved into or in front of the hard block. Therefore, we encapsulate the logic of the hard block in a second-order variable. Continuing with our DSP example, we model the block as $G(x * y + (w ? - z : z))$ where $G$ can be assigned arbitrary functions, for example $G \mapsto \lambda x.(x < c)$, meaning $G$ takes one input $x$, tests this input for less-than $c$, and returns the result of this test (Fig.4b). Another, more efficient solution would be $\{G \mapsto \lambda x.(x < 0), x \mapsto a, y \mapsto b, z \mapsto c, w \mapsto 1\}$ (Fig.4c), since tests for negativity can be done by testing a single bit when using the usual two’s complement encoding.
B. Unification Algorithm

There exist linear-time algorithms for first-order unification, which always return the most general unifier, or stop if the expressions cannot be unified [14], [15]. The operation of those algorithms can be modeled by an inference system with the pair of expressions to be unified forming the root node of a tree, and transformation rules which generate zero or more child nodes for each node in the tree. Each node consists of a set \( P \) of unsolved unification problems and a set \( S \) of variable assignments; a node with \( P = \emptyset \) contains a solution to the original unification problem in its set \( S \).

Equational unification relaxes the notion of equality between unified expressions: whereas ordinary unification seeks to find a variable assignment which makes the two expressions syntactically equivalent, a unifier in E-unification only needs to make the expressions equivalent modulo an equational theory \( E \) [15], i.e., given two expressions \( s \) and \( t \) and an E-unifier \( \sigma \), \( \sigma(s) =_E \sigma(t) \) holds, where \( u =_E v \) means that \( u \) and \( v \) are in the same E-equivalence class [16]. By adding a single rule to the inference system, the equational aspect is accounted for in the same \( E \)-equivalence class [16]. Unfortunately, this new inference system has the property that it is potentially non-terminating, i.e. that the search tree generated by the inference system is unbounded.

If the expressions of the unification problem contain higher-order variables (representing functions), the system for the first-order unification system has to be extended by three inference rules to handle the higher-order cases [17]. The extensions for equational unification and for higher-order unification are mostly independent of each other, so they can be combined to form an inference system for higher-order E-unification [18]. Fortunately, compared to general higher-order unification, our problem is simpler in that only first- and second-order variables appear, primarily because there is no need to deal with \( \eta \)-conversion [18]. Third-order (or higher) variables are not possible, since that would mean that a function itself would be an input or an output of an operator, which is not possible in hardware.

In the following, we present and explain the inference rules from [18], adapted to our needs. The initial expression system \( P; S \) consists of the unification problem \( P = \{ \langle u, v \rangle \} \), where \( \langle u, v \rangle \) is the expression pair encoding the circuit and the hard block, and \( S = \emptyset \) is the set of variable assignments. We use the notation

\[
\{ \langle u, v \rangle \} \cup P; S \rightarrow \{ \langle u', v' \rangle \} \cup P; \{ x \rightarrow w \} \cup S
\]

which means: an expression system \( P; S \) with \( \{ \langle u, v \rangle \} \in P \) generates the child node \( P'; S' \) with \( P' := P \cup \{ \langle u', v' \rangle \} \setminus \{ \langle u, v \rangle \} \) and \( S' := S \cup \{ x \rightarrow w \} \).

Trivial:

\[
\{ \langle u, v \rangle \} \cup P; S \rightarrow P; S
\]

Syntactically equivalent expressions do not need to be unified any further and can be dropped from the system. In our implementation, this rule is always applied immediately when an inference rule would create such a pair.

Decomposition:

\[
\{ \langle a(u_n), a(v_m) \rangle \} \cup P; S \rightarrow \bigcup_{1 \leq i \leq n} \{ \langle u_i, v_i \rangle \} \cup P; S
\]

where \( a \) is an arbitrary atom of arity \( n \) (i.e., a constant, a function symbol, or a variable).

Variable Elimination:

\[
\{ \langle x, u \rangle \} \cup P; S \rightarrow \sigma(P); \sigma(\sigma(S))
\]

where \( x \) does not occur as a free variable in \( u \) and \( \sigma := \{ x \rightarrow u \} \). In our case, we do not allow \( u \) to contain any higher-order variables, since together with equational transformations, this could lead to flex-flex pairs (see below). Note: the same inference rule can be applied to an expression pair of the form \( \langle u, x \rangle \).

In our implementation, this rule is always applied immediately to suitable expression pairs. Note that this excludes some solutions, because equational transformations cannot be applied to assignments in \( S \); see Section III-C for a discussion of this effect.

Imitation:

\[
\{ \langle a(m), F(n) \rangle \} \cup P; S \rightarrow \sigma(P); \sigma(\sigma(S))
\]

where \( a \) is a (function) constant or a variable and \( \sigma := \{ F \mapsto \lambda y. a(F(m, y)) \} \).

Projection:

\[
\{ \langle a(m), F(n) \rangle \} \cup P; S \rightarrow \sigma(P); \sigma(\sigma(S))
\]

for any \( i \in \{1, \ldots, n\} \), where \( a \) is a (function) constant or a variable and \( \sigma := \{ F \mapsto \lambda y_i. a(F(m, y)) \} \).

Flex-Flex:

Flex-flex pairs have the form \( \{ F(n), G(m) \} \) and lead to a potentially unlimited number of child nodes in the search tree. For symbolic refutation methods, Huet showed that resolving flex-flex pairs is not necessary, since their unification can be delayed [19]. This approach is not feasible in our case, since the higher-order variable in the hard block’s expression needs an assignment. Instead, we avoid all transformations which could lead to a flex-flex pair.

Lazy Paramodulation:

\[
\{ \langle u, v \rangle \} \cup P; S \rightarrow \{ \langle u/\beta, l \rangle, \langle u[\beta \leftarrow r], v \rangle \} \cup P; S
\]

where \( u/\beta \) means ‘location \( \beta \) in expression \( u \)’, \( u[\beta \leftarrow r] \) means ‘expression \( u \) with the subexpression at location \( \beta \) replaced by expression \( r \)’, \( u/\beta \) is not a variable and \( l \equiv r \in E \cup E^{-1} \) is a fresh instance of an equation from the equational theory \( E \). Furthermore, if \( l \) is not
a variable, then $Head(u/\beta) = Head(l)$ has to hold. This inference rule also applies to the pair $(v, u)$.

In this form, this rule would create a lot of child nodes for most nodes in the search tree. In order to avoid an excessive explosion of the search tree, we devised two modifications to this transformation. The first and most important one is that replacements are carried out at the top-level only, so that the right-hand side of the rule simplifies to $\{u/l\} \cup (r, v) \cup P; S$. Although this leads to a huge reduction in the number of generated expression systems, it most likely does not prevent the algorithm from finding an optimal solution, i.e., it will find at least all semantically different solutions (for example, it might find the assignment $x \mapsto (a + 0)$, but not $x \mapsto a$); see Section III-C.

The second restriction is due to some equations with one side consisting of a single variable (e.g., $x + 0 = x$). Instantiating this rule with $l = x$ and $r = x + 0$ would lead to the rule being applicable to both expressions in every expression pair (as opposed to $l = x + 0$ which is only applicable to expressions with the top-level symbol ‘+$’. For that reason, we oriented these rules so that their variable-only side is on the right, and instantiate rules from $E$ only. To compensate, we added the transformation with the condition that $Head(v) = Head(l)$:

\[ \{ (u, v) \} \cup P; S \implies \{ (u, r), (l, v) \} \cup P; S \quad (5') \]

We use the following laws for semantic equality (we use redundant laws to avoid long chains of transformations):

\[
\begin{align*}
x + y & \Rightarrow y + x \\
x + (y + z) & \Rightarrow (x + y) + z \\
x + 0 & \Rightarrow x \\
-x & \Rightarrow x \\
-1 \cdot x & \Rightarrow -x \\
-(x + y) & \Rightarrow -x - y \\
-(x \cdot y) & \Rightarrow -x \cdot y \\
x \cdot y & \Rightarrow x \cdot y \\
x \cdot (y + z) & \Rightarrow (x \cdot y) + (x \cdot z) \\
x \cdot 1 & \Rightarrow x \\
x \cdot 0 & \Rightarrow 0 \\
x \cdot (y + z) & \Rightarrow x \cdot y + x \cdot z \\
x \cdot y & \Rightarrow y \cdot x \\
x < y & \Rightarrow x - y < 0 \\
x < y & \Rightarrow 0 < y - x \\
s \cdot x + y : z + w & \Rightarrow (s \cdot x : z) + (s \cdot y : w) \\
s \cdot x \cdot y : z \cdot w & \Rightarrow (s \cdot x : z) \cdot (s \cdot y : w) \\
s \cdot -x \cdot -y & \Rightarrow -(s \cdot x \cdot y) \\
s \cdot x : x & \Rightarrow x \\
s \cdot x : y & \Rightarrow s \cdot y : x \\
0 \cdot x : y & \Rightarrow y \\
1 \cdot x : y & \Rightarrow x \\
!\{x\} & \Rightarrow x
\end{align*}
\]

C. Implementation

The first step of the unification process is building the initial expression system $P_0 = \{ (u, v) \}; S_0 = \emptyset$, where $u$ is the expression describing the circuit and $v$ is the expression modeling the hard block. As an invariant, the expression $v$ in an expression pair $(u, v)$ is always first-order; this way, flex-flex pairs cannot occur. This expression system is the root of the search tree generated by the inference system.

The algorithm then proceeds by choosing an unvisited node (i.e., an expression system) from the search tree. From this system $P; S$, one expression pair is chosen and child nodes are generated, one for each applicable inference rule. Note that the definition of the inference rules would actually require all pairs in $P$ to be chosen. However, it often does not make a difference whether given two pairs $p_1, p_2 \in P$, the pair $p_1$ and then the pair $p_2$ is transformed or the other way around. By selecting only a single pair for each expression system, we greatly reduce the search space. As a metric for choosing a pair, we use the maximum depth of the two expressions it contains – this way, ‘simpler’ pairs should be unified first, thereby generating variable assignments which hopefully simplify the more complex pairs.

The search tree is processed in a combination of depth-first and breadth-first search: given an expression system, the search subtree generated by it is traversed up to a certain depth; expression systems which at this point are not solved yet are added to a queue which represents the ‘search front’. Expression systems which are solved (i.e., $P = \emptyset$), are added to the list of solutions. For the depth-first search, a function parameter indicates the remaining recursion depth: a value of 0 means that this expression system should be added to the queue, a value of 1 means that one level of transformations is left, and so on. After each transformation, this value is decremented by 1 with the following exceptions:

- Decomposition never decreases the remaining depth. Even though this inference rule usually increases the number of pairs, it simplifies the expressions. In practice, this optimization has lead to reduced runtimes.

- Imitation and projection rules do not decrease the remaining depth if their application has lead to an expression system with fewer pairs.

We have found that an initial value of 1 leads to the best results (see Section IV). Furthermore, we have implemented the following optimizations:

- For pairs of ground expressions (i.e., those that do not contain any variables), we use the SMT solver Yices\(^2\) (version 2.1.0) to check for semantic equality directly.
- To do this, we build an assertion of inequality between the two expressions; if this assertion is unsatisfiable, the expressions are equal and do not have to be unified further; if it is satisfiable, the expressions cannot be unified (due to the lack of variables) and the expression system is thrown away. As a consequence, expression pairs like $\{0, 1\}$ are removed to free resources for other candidate expression systems; this would not be possible otherwise, since E-unification

\(^2\)http://yices.csl.sri.com
can only show equality, not inequality. Tested pairs, along with their results, are stored in a hash table in order to reduce redundant computations.

- The search front grows exponentially with the depth of the traversed search tree (because most nodes produce several child nodes), so that reaching the next level of the tree takes exponentially more time. Most of the paths through the tree, however, are going in the wrong direction and add more complexity to the expression pairs without actually getting closer to a solution. We therefore devised a simple heuristic: whenever more than a certain number of nodes are in the queue, the most complex expression systems are discarded; the complexity of an expression system is defined here as the maximum depth of its expressions. We found that a threshold of 50,000 and removing half of the nodes is a good compromise in practice.

- When decomposing if-then-else expressions and the ‘if’-expression of one of them is a constant 0, then the ‘then’-expressions of both are discarded. Conversely, if one of them is a constant 1, then the ‘else’-expressions are discarded.

- All generated expression systems are stored in a hash table. Hence, if the (syntactically) same expression system is generated again, it is not added to the queue of unvisited nodes.

- Similarly, equation rule instantiations for a given expression pair are cached. This is necessary since otherwise, nearly all expression systems generated through lazy paramodulation would be syntactically distinct and the optimization above would be ineffective.

No evaluation with respect to area or delay of the solution is done by our implementation: this process depends on the FPGA in question and is independent of our unification algorithm, so it is better done afterwards by a third-party tool.

In many cases, the algorithm returns a solution which contains an unoptimized assignment, e.g. \( x \rightarrow (a + 0) \) or \( y \rightarrow (b + c + 1) \). These results are ‘artefacts’ of the inference rule applications, and sometimes no better version of these solutions is found. There are two reasons for this:

1) Once an assignment for a variable is found, it is no longer subject to equality transformations (which could simplify \( a + 0 \) to \( a \)).

2) Equality transformations are applied to the top-level of the expressions only, hence \( b + c + 1 \) cannot be simplified to \( b + c \).

Although it would be possible to generate optimized solutions from the imperfect ones by randomly applying transformation rules to assignments, we decided against this inefficient approach since it would produce many more suboptimal solutions. Instead, this simplification can be handled by the FPGA synthesis software.

IV. RESULTS

We tested our implementation on a system with a dual-core Intel Core i5 660 CPU at 3.33 GHz and 8 GiB memory using Mono version 2.10. Since our algorithm successively lists all solutions it finds, we measured the time it took to find an ideal solution (respectively one which could be easily optimized to be an ideal solution, as explained in Section III-C).

The most important hard block type with combinational functionality on current FPGAs are DSP blocks, hence we concentrated on them. Table I lists the circuits we used as benchmarks, along with an optimal solution (i.e., one which requires the least amount of additional logic). In Table II, we present the results from applying our algorithm to several sample circuits for the following hard block \( K \), which models the most important combinational subset of the Xilinx Virtex-5 DSP element.

\[
K := s_1 \land t_1 : t_0
\]

where

\[
\begin{align*}
t_0 &:= (C + t_2) + c_{in} \\
t_1 &:= C + \neg (t_2 + c_{in}) \\
t_2 &:= s_2 \land (A_1 \land A_2) : A_0
\end{align*}
\]

Since higher-order E-unification is inherently non-terminating, we stopped the execution after 10 seconds because it was unlikely that better assignments would be found after that time. For some of the circuits, our algorithm did not find the optimal solution for all combination of the parameters; it did, however, usually find several that were close to optimal.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.2</td>
</tr>
<tr>
<td>3</td>
<td>3.5</td>
</tr>
<tr>
<td>4</td>
<td>3.6</td>
</tr>
<tr>
<td>5</td>
<td>2.8</td>
</tr>
<tr>
<td>6</td>
<td>3.1</td>
</tr>
<tr>
<td>7</td>
<td>4.6</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4.6</td>
</tr>
<tr>
<td>12</td>
<td>2.0</td>
</tr>
<tr>
<td>13</td>
<td>2.2</td>
</tr>
<tr>
<td>14</td>
<td>3.4</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE I. BENCHMARKED CIRCUITS.**

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.6</td>
</tr>
<tr>
<td>3</td>
<td>0.7</td>
</tr>
<tr>
<td>4</td>
<td>0.8</td>
</tr>
<tr>
<td>5</td>
<td>0.9</td>
</tr>
<tr>
<td>6</td>
<td>1.0</td>
</tr>
<tr>
<td>7</td>
<td>1.1</td>
</tr>
<tr>
<td>8</td>
<td>1.2</td>
</tr>
<tr>
<td>9</td>
<td>1.3</td>
</tr>
<tr>
<td>10</td>
<td>1.4</td>
</tr>
<tr>
<td>11</td>
<td>1.5</td>
</tr>
<tr>
<td>12</td>
<td>1.6</td>
</tr>
<tr>
<td>13</td>
<td>1.7</td>
</tr>
<tr>
<td>14</td>
<td>1.8</td>
</tr>
<tr>
<td>15</td>
<td>1.9</td>
</tr>
<tr>
<td>16</td>
<td>2.0</td>
</tr>
</tbody>
</table>

**TABLE II. EXPERIMENTAL RESULTS FOR HARD BLOCK K. NUMBERS ARE TIME IN SECONDS; D IS THE TRAVERSAL DEPTH, +Y IS WITH YICES, -Y WITHOUT IT. ‘---’ INDICATES THAT THE OPTIMAL SOLUTION WAS NOT FOUND.**
It can be seen from the benchmark results that a traversal depth of 2 generally yields shorter runtimes, but does not find as many optimal solutions as a traversal depth of 1. To our surprise, using Yices for testing equality of ground pairs actually slows down the algorithm slightly, which indicates that our equational unification implementation works very well for proving semantical equality. Nevertheless, in at least one case, Yices was necessary to find the best solution.

One circuit failed to be mapped to the hard block for any set of parameters, even though a sequence of transformations to a solution exists. We still have to investigate why the optimal mapping was not found, but we suspect that its intermediate expression system became too complex and hence was discarded. A related problem we noticed during testing is that, often, a suboptimal proto-solution (i.e., one which has a suboptimal assignment for some of its inputs, leaving only non-complex expression pairs to be unified) is found quickly and, in turn, spawns many child nodes with slight variations, which displace more promising (but temporarily more complex) expression systems. We also found that the higher-order part of the unification process does not take a lot of time, so removing the ability to add logic to the hard block’s output would not be very beneficial.

V. CONCLUSION

In this paper, we present a solution to the hard IP block instantiation problem for FPGAs, building upon higher-order E-unification. We provided all important details of our implementation and reported about experimental results which demonstrate the potential of this technique. The current state of our algorithm is promising and can already find the optimal hard block instantiation for many non-trivial circuits. The time required to find an optimal solution is quite short, making this approach interesting for interactive optimization and design space exploration.

A. Future Work

As mentioned in Section IV, the biggest problem to increase the efficiency of our algorithm are the expression systems which have already found a suboptimal assignment for some of the inputs. Due to the low complexities of their remaining expression pairs, they are preferred by the unification algorithm, thereby producing many similar, suboptimal solutions and displacing other transformation paths to better solutions. It turned out to be quite difficult to recognize these situations algorithmically.

We will also work on generalizing our approach to sequential circuits and hard blocks, so that hard block registers can be used for pipelining and internal feedback paths can be used for implementing accumulators. This will also result in more complex hard block models, since many control inputs and internal multiplexers are typically used for sequential circuits.

REFERENCES


