Intelligent Hardware Compilation with Options The CCC HLS system and XML schema

Michael Dossis

Department of Informatics Engineering, TEI of Western Macedonia Kastoria Campus, Fourka Area, Kastoria, GR 52 100, Greece

Accepted 2nd February, 2015

Abstract

The increasing complexity of modern ICs has motivated research in high-level and system synthesis (HLS and ESL). Formal, rapid and intelligent HLS techniques are discussed in this work, that make the generated implementations correct-by-construction. Our intelligent techniques include compiler-compiler, RDF (Resource Description Framework) and logic rules, along with a number of synthesis options that are utilized by the CCC hardware compiler. The presented tools utilize compiler-generators, RDF rules and logic programming in combination with XML validation of the internal state of the hardware compiler. These intelligent and formal techniques make the whole transformation from source code to implementation, formal and the generated implementations provably-correct. The CCC tool is enhanced with the Parallel, Abstract Resource - Constrained Scheduler, which aggressively optimizes the state schedules, into maximally parallelized ones. A number of options are applied to the CCC HLS tool, in order to automatically compile selected test cases from real-world applications that are discussed at the end of this work and which prove the usability of the formal tools and our intelligent HLS compiler.

Keywords: Microelectronics design; High level synthesis; Formal languages; Electronic design automation; Scheduling; Compilers; RDF; Formal methods; Logic programming; Custom microarchitecture

Introduction

Digital microelectronic ICs that are found in embedded, high-performance and portable computing systems, nowadays have highly complex design hierarchy, control flow and interconnections. During the last couple of decades, commercial and academic organisations have invested in HLS and optimisation techniques, so as to achieve design automation, quality of implementations and short specification-to-product times (Gal, 2008), (Gupta, 2004).

However, existing HLS tools are not widely accepted by the engineering community because of their poor results, especially for large applications with complex module and control-flow hierarchy. Very often, the programming style of the source code has a severe impact on the quality of the synthesized implementation. For large-scale applications, the complexity of the synthesis transformations (front-end compilation, algorithmic transformations, optimizing scheduling, allocation and binding), increases exponentially, with a linear increase of the design size (Walker, 1995). Existing HLS tools impose proprietary extensions or restrictions (e.g. exclusion of while loops) on the programming model of the specifications that they accept as input, and various heuristics on the HLS transformations that they utilize (e.g. guards, speculation, loop shifting, trailblazing). Most of them are suitable for linear, dataflow dominated (e.g. stream-based) designs, such as pipelined DSP, image processing and video/sound streaming.

The contribution of this work is an integrated HLS toolset which utilises intelligent and formal techniques so as to apply the source-to-implementation optimization transformations, thus, the produced hardware implementations are correct-by-construction. Therefore, the design needs verification only at the top behavioral level, without spending days or even weeks, on lengthy RTL or annotated gate simulations. Moreover, various custom options can be applied by the user on the automatic HLS transformation, such as the type of the micro-architecture, the generated HDL code as well as the inclusion of custom (e.g. arithmetic) logic functions throughout the HLS compilation.

The author has designed and developed (Dossis, 2011), an intelligent HLS compiler that includes a scheduler of operations into control steps, achieving the maximum functional parallelism in the synthesized implementation (Paulin, 1989). This HLS scheduler called PARCS, utilizes logic programming (Nilsson, 1995) and RDF subject-predicate-object relations (Allemang, 2011), to formally achieve the maximum possible parallelism of operations. In this way, the functionality of the delivered implementations is correct-by-construction. (Walker, 1995) explores various scheduling techniques.

Formal HLS techniques are analysed in the next section. Next, the intelligent approach of the prototype optimising CCC synthesizer is described, such as formal predicate logic (Nilsson, 1995), RDF relations and XML schema validation. Then, the usability and correctness of the CCC HLS toolset are evaluated with a number of benchmarks. The last section draws useful conclusions and proposes future work.

Existing Work in intelligent HLS techniques

Established and well studied HLS tasks include scheduling, allocation and binding (Walker, 1995). The front-end part of HLS tools include parts of software programming language compilers (Holub, 1990), such as parsing, semantic
analysis, intermediate variable optimization, elimination of
dead code, etc. The front-ends exchange information with the
back-ends using intermediate formats, such as the
Electronic Design Interchange Format (EDIF) (Wikipedia,
2012), (Rubin, 2012), used by most E-CAD tools. Complex
control flow optimization has been evaluated in (Kountouris,
2002), (Gupta, 2004), (Wang, 2003), but for small parts of
code and by no means complete application tests.

(Huang, 2007) discusses synthesis for distributed logic and
memory. (Huang, 2003) uses communicating processes as a
system specification medium. HLS methods that include
memory access management are outlined in (Gal, 2008),
where digital signal processing (DSP) and streaming
applications are synthesized using performance constraints.
(Gupta, 2003) analyses mutually exclusive scheduling on
extended data-flow graphs (EDFG). (Molina, 2009)
synthesizes behavioural descriptions with time constraints,
where complex operations are decomposed into simpler
ones, and a similar set of decomposed fragments of
operators, with the same pattern, are scheduled in a clock
cycle.

In (Keinert, 2009) an actor, that is used to model every
module or system process, communicates with other actors
via communication channels. These actors are used by the
SystemCoDesigner (Keinert, 2009) to exercise electronic
system level (ESL) design space exploration. In (Kundu,
2010) the SURYA system utilizes the Simplify theorem
prover to prove that the RTL model generated by HLS tools
is functionally-equivalent to the specification. SURYA found
two bugs in the SPARK HLS tool (Gupta, 2004), which were
until then unknown. In (Paik, 2010) flip-flops are replaced
with latches so as to improve implementation timing, since
latches are inherently more tolerant to process variations
than flip-flops. Nevertheless, latch-based design is more
cumbersome than flip-flops.

The W3C Resource Description Framework

The Resource Description Framework (RDF) is a metadata
model and is used to model the information of web resources
(Allemang, 2011). RDF models include subject-predicate-
object relations with explicit statements, called triplets.
Triplets specify resources, that store knowledge, information
and data retrieval in large automated software
tools. This is achieved due to the suitability of RDF to
capture, store, exchange, and use machine-readable web
information. Here, RDF is used to formally model and
validate the internal data and state of the author’s HLS tool.

The eXtensible Markup Language (XML) serialization format
is used to define RDF. XML’s syntax is formally specified
(Allemang, 2011) and is ideal to model simple triples such as
subject-predicate-object (and other) relations. The following RDF relation:

```xml
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:dc="http://purl.org/dc/elements/1.1#"
 xmlns:Description rdf:about="http://www.awl.com/Formal_Synthesis">
  <dc:title>High-level Synthesis</dc:title>
  <dc:publisher>Addison-Wesley Publishing</dc:publisher>
</rdf:RDF>
```

A valid XML instance can be checked against the rules of an
XML schema. This is done by certain XML (commercial or
free) parsers compatible with specific XML schema
implementations such as the DTD or the Relax NG languages.
The following XML instance is automatically produced by
the author’s HLS tool (see following paragraphs), and it
defines a data type and two subprograms in the tool’s source
ADA programs:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Produced by CCC front-end compiler -->
<schemaxmlns=http://www.w3.org/2001/XMLSchema
targetNamespace="http://www.w3.org/2001/XMLSchema">
  <annotation>
    <documentation>
      XML schema for a hierarchical module of the source code
    </documentation>
    <annotation>
      <complexType name="hierarchy_part">
        <sequence>
          <element name="type_def_natural_2048"/>
          <sequence>
            <element name="data_object_variable2"/>
            <element name="data_object_constant1_value_100"/>
            <element name="data_object_constant1_value_1000"/>
          </sequence>
          <element name="function_convert3"/>
          <sequence>
            <element name="input_parameter_my_input1"/>
            <element name="input_parameter_my_input2"/>
            <element name="input_parameter_my_input3"/>
          </sequence>
          <element name="procedure_differential2"/>
          <sequence>
            <element name="input_parameter_my_input4"/>
            <element name="input_parameter_my_input5"/>
            <element name="output_parameter_my_output1"/>
          </sequence>
        </sequence>
      </complexType>
    </annotation>
  </schemaxmlns=http://www.w3.org/2001/XMLSchema>
</schema>
```

The above XML instance defines the data type
"natural_2048" and two subprograms, the function
"convert3" with three formal input parameters "my_input1",
"my_input2" and "my_input3" as well as procedure (see
ADA-95 definition) differential2 with two formal input
parameters and one output. These elements are of type
"hierarchy_part", and each one of them includes a sequence of "leaf" sequences with the structure of all child elements
with type defined above. The graphical validation result of
the above XML code is shown in Fig. 1, and it was
automatically produced by a DTD parser that helped to
identify and correct some initial semantic issues in the tool
development.
FIG. 1 VALIDATION OF AN XML DOCUMENT INSTANCE AGAINST A SPECIFIC XML SCHEMA (AUTOMATICALLY PRODUCED FROM INTERNAL COMPILER DATA)

As widely documented, XML is a formal way to model internal information in design (E-CAD) tools. XML representation of C code is introduced in (Atsumi, 2011) to aid the CASE tool development. A hash table and a stream index are used in (Weijian, 2011) to filter out invalid elements of an XML document. The IEEE std 1685-2009 IP_XACT, is based on XML schema, and is used as internal representation and exchange format between EDA tools (IEEE IP-XACT std, 2010). Such internal data include design units, interconnections, functional primitives, metadata, and IP blocks.

XML schema and the eXtensible Stylesheet Language Transformations (XSLT) language are used in (Jumaa, 2010) to enable the sharing, conversion, transfer and exchanging of healthcare database data. Many academic/commercial database and web tools utilise XML schemas and instantiations to represent internal information, such as primary data generated automatically and used by database, multi-media and web processing tools.

XML is combined with Web technology in (Sanchez-Martinez, 2010) to structure, consult and share corporate data. Similarity algorithms for XML documents are analysed in (Sun, 2010). The IBM DB2 query matching and compensation techniques are enhanced with XML functionality, to implement and evaluate query rewrite rules in (Godfrey, 2009). An incremental approach called “T-Schema” (of XML-to-relational mapping storage), is proposed in (Xu, 2010), to address the strong dynamics of XML. The SMOQE tool (Fan, 2007) generates the first regular XPath engine and provides answering queries technique, over recursively defined XML views. (Rajugan, 2006) discusses the use of semantic web for EIS and databases. A XML metamodel captures NFRs and their relations in (Kassab, 2008).

The Intermediate Predicate Format

The Intermediate Predicate Format (IPF) was invented by the author of this paper, to model the design and the HLS transformations in the CCC HLS tool. (Dossis, 2010) analyses the syntax and semantics of IPF, which uses the resolution of Horn clauses as formal object relations (Nilsson, 1995), to implement the HLS transformations. The front-end phase of the CCC compiler (see following paragraphs), generates the IPF database to capture all the algorithmic, structural and data typing attributes of the source program, as in the following Prolog fact:

```
fact_id(object1, object2, ..., objectN) (form 1)
```

The Prolog predicate name fact_id relates the objects object1 to objectN in a formal manner. The identifier fact_id names the logical relation between the above objects. IPF facts represent program operations, data object descriptions, data types, operators, subprogram calls, etc. The back-end phase of the CCC compiler applies HLS transformations on these facts so as to produce the optimized design implementation. This is done in combination with the internal logical rules as a design knowledge-base in order to "conclude" and infer the RTL (register-transfer level) implementations.

The IPF syntax facilitates declarative processing by Prolog predicates, as well as imperative processing. This is because IPF facts (e.g. data table facts) are referenced with their entry numbers in other IPF facts (e.g. program table facts).

The IPF XML schema view is used to validate the internal state of the front-end and back-end phases of the CCC HLS tools. The following XML instance models the program statement (Dossis, 2010) of form 1:
The Intelligent CCC HLS Techniques

The synthesis design flow

The HLS flow includes two major steps: the front-end phase and the back-end phase. These two tools exchange data via the IPF database. XML instances of IPF are automatically generated to validate the compilation process, using graphical or command-line based XML validators. The front-end phase performs the typical tasks of a software compiler, such as parsing, generation of abstract syntax trees, type-checking of data and program statements, optimization of auxiliary variables and constants, generation of syntactic and semantic error messages etc.

The XML schema view is also used by the front-end and back-end phases to formally validate the translation, since IPF constitutes a formal link between the two phases of the hardware synthesizer.

CCC analyses the IPF database and generates the initial FSM schedule, considering custom options (such as the location of large data objects on shared memory). Then, it optimises the initial schedules using the PARCS (Parallel Abstract Resource - Constrained Scheduling) scheduler. PARCS works with, or without resource constraints, and generates the maximally parallel hardware implementation (Dossis, 2011), while satisfying source code dependencies.

A custom options file can be used to mark certain program subroutines as "custom blocks", so that they are used as pre-optimized and static custom modules. Hardware arithmetic blocks, or complete data-flow systems such as DSP filter blocks and cryptographic mathematical functions, can be used as custom blocks as in the benchmarks analysed in later sections.

The validation of this XML instance is shown in Fig. 2.

Other custom options of the back-end compiler are the targeting of either massively-parallel (with resource redundancy), or conventional FSM+datapath micro-architectures modelled in Hardware Description Language (HDL) RTL code. More options include the targeted language which (at the moment) include VHDL and Verilog HDL.

Formal Back-end HLS Transformations The logical relations of the back-end compiler use definite clauses (Nilsson, 1995) such as follows:

\[ A_0 \leftarrow A_1 \land \ldots \land A_n \quad (\text{form 2}) \]

where \(\leftarrow\) is the logical implication symbol (\(A \leftarrow B\) means that if \(B\) applies then \(A\) applies), \(\land\) is the logical conjunction symbol, and \(A_0, \ldots, A_n\) are atomic formulas (logic facts) of the form:

\[ \text{predicate}_\text{symbol}(\text{Var}_1, \text{Var}_2, \ldots, \text{Var}_N) \quad (\text{form 3}) \]

where the positional parameters \(\text{Var}_1, \ldots, \text{Var}_N\) of the above predicate "\(\text{predicate}_\text{symbol}\)" are either variable names (such as in the back-end inference rules), or constants (such as in the IPF table statements) (Nilsson, 1995). By combining these, the source code subroutines are transformed into optimized, provably-correct RTL hardware implementations.

Formal validation using XML schema

Formal logic rules (logic relations) as in form 2 construct the back-end inference engine. Hence, the IPF's facts "drive" the logic rules of the back-end compiler which infers provably-correct hardware implementations, which are technology-independent, free of any standard template, and custom microarchitectures in synthesizable HDL code.

The PARCS optimizer works on the enhanced with the custom user options schedule, such as the shared memory.
access operations. The XML view is validated for the intermediate representations and processes throughout the various phases of the compilation. Here follows the XML schema validation of the state(...) inference rule:

```xml
<complexType name="state">
  <sequence>
    <element name="Module_1"/>
    <element name="parcs"/>
    <element name="parcs_state_number"/>
    <element name="parcs_state_name"/>
    <element name="parcs_next_state"/>
    <element name="no_conditional_transition"/>
    <element name="scheduled_operation_list">
      <complexType>
        <sequence>
          <element name="Operation_1"/>
          <element name="Operation_2"/>
          <element name="Operation_3"/>
        </sequence>
      </complexType>
    </element>
    <element name="no_conditional_operations"/>
  </sequence>
</complexType>
```

This XML instance models one PARCS state of design Module_1, with three scheduled operations in parallel and with no conditional operations or transitions. Both the logic and the XML views of IPF are extracted automatically by the front-end and back-end compilation phases, and they are validated in both logic programming and XML views. The graphical validation of the above XML PARCS state instance is shown in Fig. 3.

The XML view of the HLS transformations and data of the back-end compiler consists of relations between predicate symbols. An example logic programming view of the relation transform1 (HLS transformation) between objects operation1, operand_A, operand_B, result_C is the following:

```
th:transform1(operation1, operand_A, operand_B, result_C).
```

Here follows the XML view of the same relation:

```xml
<complexType name="transform1">
  <sequence>
    <element name="operation1"/>
    <element name="left_operand_A"/>
    <element name="right_operand_B"/>
    <element name="result_data_C"/>
  </sequence>
</complexType>
```

The graphical XML validation of the same transformation is shown in Fig. 4. The XML schema instances are validated using any available validators.

---

**FIG. 3** XML SCHEMA VALIDATION OF A PARCS STATE

**FIG. 4** XML SCHEMA VALIDATION OF THE TRANSFORM1 BACK-END COMPILER PREDICATE

---

*How to Cite this Article:* Michael Dossis "Intelligent Hardware Compilation with Options: The CCC HLS system and XML schema" Weber Engineering and Technology, Vol. 1 (1) 2015, Article ID wet_111, 62-70, 2015
Experimental Designs with Custom Options

A large number of ADA designs were synthesized with the CCC compiler and all of them were proven by simulations to be accurate and matching the behaviour of their source models. Here two of them have been analysed in order to demonstrate the usability of the CCC HLS custom options. These are a DSP FIR filter, and a RSA crypto-processor from cryptography applications.

In all tests, the intermediate form and the internal HLS transformations were validated against the respective XML schemas which were automatically extracted from critical points in the compilation flow. For the DSP filter, the two lower-level subroutines in ADA, which model the processing of one more incoming sample and the shifting of the filter history by one position, were chosen to be implemented as custom combinatorial blocks taking one clock cycle to implement.

A top-level subroutine which contains calls to the above subroutines, was used to process a whole length of incoming sample arrays.

![FIG. 5 FIR FILTER ADA CODE HIERARCHY](image)

This subroutine was processed normally via the CCC compiler and produced VHDL code with an optimized FSM of 10 states. Within appropriate state descriptions of this FSM, the above custom block subroutines are called via VHDL call mechanism. The whole ADA coding and compilation of the DSP filter took less than half an hour to run. The hierarchy of the FIR ADA code and implementation is shown in Fig. 5.

The experimental CCC environment is shown in Fig. 6. The intended system is modelled in the ADA programming language. The initial ADA specification model is compiled with the GNU ADA and the generated binaries are verified with the ADA testbench and test vectors. This exhibits a rapid verification manner, due to extremely high compile-and-execute verification speed. Then the ADA subroutines that are intended for hardware (microelectronic) implementation are integrated in an autonomous ADA package (library module). The latter is compiled and synthesized into hardware using the prototype CCC tools. The generated hardware modules are downloaded in an Xilinx Virtex-2 FPGA was accommodating the synthesized hardware blocks, with the synthesis flow and the target architecture shown in Fig. 6.

The initial state schedules are first extended with custom option - guided operations and the result is optimized with the PARCS scheduler as a new schedule. Statistics regarding the optimization rates of the PARCS scheduler are shown in Table 1.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Initial Schedule States</th>
<th>PARCS Result States</th>
<th>State Reduction Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP FIR filter processor</td>
<td>17</td>
<td>10</td>
<td>41%</td>
</tr>
<tr>
<td>RSA cryptoprocessor</td>
<td>16</td>
<td>11</td>
<td>31%</td>
</tr>
</tbody>
</table>

¹The intermediate predicate format is patented with patent number: 1006354, 15/4/2009, from the Greek Industrial Property Organization
²This hardware compiler method is patented with patent number: 1005308, 5/10/2006, from the Greek Industrial Property Organization
Table 1 indicates that the states reduction rate reaches up to 41% for the FIR processor case. This is a very efficient hardware implementation of large designs with a few hundred states per module, such as large ASICs and complex IP modules being part of embedded system SoCs.

The code of the ADA subroutines can be either standalone or hierarchical. This means that a number of ADA subroutines can include function/procedure calls to other subroutines of the library that is synthesized. This hierarchy is maintained through the CCC compilation. Thus, the CCC designer dictates the modularity of the generated hardware blocks. Additionally, all the necessary co-processor interfaces and (e.g. memory) communication protocols are automatically synthesized by the CCC compiler and inserted in the initial schedules derived directly from the input models. This is implemented via a memory custom options file. Moreover, selected ADA subroutines can be “marked” as custom arithmetic modules. These are usually complex Boolean functions and they are inserted as expanded VHDL procedure calls in the generated FSM states, executed in a clock cycle. The custom blocks option strategy was followed on the lower-level functions of the FIR DSP filter and RSA crypto-processor benchmarks.

Table 1 demonstrates that the number of hardware states is increasingly high, which constitutes the importance of the CCC framework contribution, in keeping the complexity of contemporary designs under control. It is a shared experience by experienced hardware designers, that development and verification of very complex FSMs with over 20-30 states is cumbersome and extremely prone to errors. Experienced programmers can use the CCC toolset, to implement very complex hardware designs in a few hours, whereas this takes usually more than 6 months of traditional development and verification time. This is due to the use of formal logic programming and RDF validation techniques embedded in the intelligence of the CCC synthesizer.
The ADA code hierarchy for the RSA public-key cryptography application is shown in Fig. 7. The modular exponentiator and multiplier automatically produce optimized FSMs with 12 and 29 states respectively. The other 3 lower-level modules are transformed as custom blocks and their VHDL calls are instantiated into the higher-level FSM state descriptions, as shown in Fig. 7. The multiplier subroutine is called within the exponentiator subroutine. This is a special case for the CCC translation and is being dealt as such by the compiler. In particular, this subroutine call is used to generate an "interface event" between the modular multiplier module and the exponentiator module. In general, when across both sides of a subroutine call, none of them is intended to be a custom block, then the subroutine calling is converted into a HDL module interface and data exchange mechanism. All of the VHDL modules for the FIR filter and the RSA cryptoprocessor have been simulated (although due to the formal nature of the CCC tool, not necessary) and the results coincided with the results of the ADA verification testbenches.

Conclusions and Future Work

The main contribution of this work is a formal, high-level hardware synthesis toolset and HLS method which were developed by the author of this paper. The CCC HLS tool utilizes compiler-compiler and logic inference techniques to turn synthesis formal. The synthesis transformations are enhanced with XML schema validation as well as RDF logic relations to implement the execution and formal validation of the internal state of our prototype hardware compiler. XML views of the intermediate format (IPF) as well as of the formal CCC compiler transformations are validated against their schema views. Thus the whole synthesis path from specification (C or ADA) to implementation (scheduled RTL) is formal and the generated implementations provably-correct.

Arbitrary and general input C/ADA code is synthesized into functionally-equivalent RTL VHDL/Verilog hardware implementation. We synthesized many applications using the CCC tools, and two, very indicative ones of them were presented and discussed in this paper. In any case, the functionality of the produced hardware accelerators (coprocessors) matched that of the input subprograms. This was anyway expected due to the formal nature of the CCC transformations. The synthesized hardware is used to accelerate time-critical parts of complete hardware/software systems. The PARCS scheduler achieves high state-optimization rates which exceeded 36% in some cases (see Table 1).

Future work includes more input programming languages (e.g. Matlab, SystemC,C++) and a more globalized use of RDF techniques throughout the flow of the HLS toolset. Moreover, more diagrammatic system modelling formats are explored such as the UML diagrams to play the role of system hardware accelerator models.

References


