Automatic Generation of Massively Parallel Hardware from Control-Intensive Sequential Programs

Michael F. Dossis
Department of Informatics and Computer Technology
TEI of Western Macedonia
Kastoria Campus, GR 52 100 Kastoria, Greece
mdossis@yahoo.gr

Abstract—High-level synthesis has been envisaged as a suitable methodology to design and deliver on time, at least large parts of today’s complex IC systems. This paper describes a unified and integrated HLS framework, to automatically produce custom and massively-parallel hardware, including its memory and system interfaces from high-level sequential program code. Using compiler-generators and logic programming techniques, provably-correct hardware compilation flow is achieved. The utilized hardware optimization inference engine is driven by a set of resource constraints, which limit to a certain boundary the number of available hardware operators to function in parallel during each control step. This optimization reduces drastically the number of different control steps (states) of the implemented application. The hardware compilation runs are completed in orders-of-magnitude less time than that which would be needed by even very experienced HDL designers to implement the same applications in RTL code. Implementation results from synthesis of a number of control-dominated, linear and repetitive, applications including a MPEG video compression engine with up to a few hundred states, are presented. In all cases the HLS framework delivers quickly provably-correct, implementable RTL code and the optimized schedule is reduced at up to 30% in comparison with the initial schedule.

Keywords—ESL, High-Level Synthesis, E-CAD, FSM, compilers

I. INTRODUCTION

Nowadays, silicon technology allows the assembly of extremely complex systems into a single IC or a small set of ICs. Applications and products of this type are found in embedded devices, multimedia and consumer services, medical and scientific data processing, telecommunications, banking, aerospace, naval and transport infrastructure.

The challenge of quickly delivering complex optimized hardware and systems has forced industry and academia to invest in rapid methodologies to achieve greater system design and development automation [1]. One such methodology is high-level synthesis (HLS) which is becoming indispensable, by utilizing techniques from the regular program compilers, and by offering the capability of electronic system-level (ESL) design to a number of popular E-CAD tools.

The contribution of this paper is an integrated and automated formal methodology and tools to rapidly deliver customized hardware, massively-parallel architectures which are tailored to specific functions and thus increase the performance of the host computing system [1]. The produced hardware includes all the necessary addressing, communication and interfacing with local or external memories. The tools are suited to processing program code with complex control flow. It is argued here, that the presented methodology reduces the specification, design, verification and implementation cycles of computing products by orders of magnitude, due to the formal and automated synthesis techniques which are employed.

The author developed a prototype set of hardware synthesis tools1. The tools include the front-end compiler, the intermediate format and the back-end compiler. The Formal Intermediate Format (FIF)2 is encapsulated in logic predicate clauses and the back-end compiler uses the resolution of Horn clauses as the building blocks of a hardware mapping and optimizing inference engine [2].

The synthesis method, as well as results from experimental hardware compilations are evaluated. Section II discusses related work. Section III introduces the hardware compilation flow. The design and verification flow are discussed in section IV. Section V outlines the inference engine and the optimizing transformations. In section VI the type, structure and interfaces of the generated hardware are introduced. Experimental results and statistics are discussed in section VII. Finally, section VIII draws conclusions and proposes future extensions to this work.

II. RELATED WORK

An early technique to synthesize algorithmic DSL code into hardware implementations is reported in [2]. Using proprietary specification formats, and targeting specific applications domain (e.g. DSP) or hardware architecture templates is found in [4], [5], and [6]. High-level synthesis tasks such as the scheduling optimizations have been studied in [1], [7], [8], [9]. Although there are nowadays a few commercial and research hardware compilation tools, most of them are suited to process linear data-flow dominated designs. The presented synthesis

1 This hardware compiler method is patented with patent number: 1005308, 5/10/2006, from the Greek Industrial Property Organization

2 The formal intermediate format is patented with patent number: 1006354, 15/4/2009, from the Greek Industrial Property Organization
framework can process any arbitrary program code sets with as much complex control flow. The program code can be also hierarchical with as many subroutines calling other subroutines.

The back-end phase of the hardware compiler is the HLS core of the framework and it transforms the generated FIF database into a set of standalone hardware FSMs which are functionally equivalent to the source code subprograms. The back-end compiler produces also state measurement statistics for the initial (unoptimized) as well as the optimized schedules.

III. SYNTHESIS FLOW

The formal intermediate format (FIF) constitutes the information exchange link between the front-end and the back-end parts of the hardware synthesizer as shown in Figure 1.

The user of the flow in Figure 1 can extend the set of accepted source program data types and operators, by updating the library file (coded in a FIF syntax subset), and by integrating these updates via the use of a library compiler. During synthesis, all the arbitrarily-complex control flow of the source programs is analyzed into a number of simple if-then-else constructs which in turn are mapped into conditional state transitions in the generated FSM+datapath. The generated FSM+datapath along with the necessary interfaces, constitute syntactic and semantic error messages, production and processing of abstract syntax trees, type-checking of data declarations and program statements, optimization of auxiliary variables and constants etc.

The front-end compiler formally transforms a set of source code subprograms into the FIF (logic clause) database file. This file captures the typing, structural, hierarchy and algorithmic (control & data flow) information of the source code. The front-end compiler performs the typical tasks of a program compiler such as lexical and syntactic analysis, generation of
the generated co-processors, which act as system accelerators in the host environment.

IV. DESIGN AND VERIFICATION

The proposed methodology targets applications that may consist of a mixture of hardware and software components. Every application is specified and coded in a set of high-level program subroutines. The hardware partition of the application is done by simply putting together a set of subroutines to be synthesized into equivalent hardware co-processors. This partition is placed in a separate library module (package) with code of sequential ADA, Pascal or behavioural VHDL. Rapid prototyping of the software components is achieved via regular software compilation and binary execution techniques of the application. Similarly rapid prototyping of the hardware components is achieved by using the compilation flow, presented in this paper.

Verification of the produced application can be done at various abstraction levels of the design representation:

1. At the abstract, algorithmic program code level (behavioural hardware model subroutines) which features the fastest verification cycle (compilation & execution).
2. At the generated synthesizable RTL HDL hardware model level, which is as fast as technology-independent RTL verification (e.g. VHDL simulation).
3. At the hardware (gate) netlist level, which is generated by established industrial or research RTL synthesizers, and it is as fast and precise as technology-specific gate-level simulations.

Figure 2 emphasizes the capacity to execute hierarchical verifications at different abstraction levels from the high-level algorithmic program code down to the detailed, technology-specific hardware implementation. It is well established that verifications at the executable program code level are orders of magnitude faster than detailed hardware technology implementation levels such as gate and device level. It is also recognized that high-level program execution is much faster than cycle-accurate (RTL) code simulations [19].

V. INFERENCE LOGIC AND BACK-END TRANSFORMATIONS

The back-end compiler is built using logic programming techniques. It reads and incorporates the FIF statements into its internal knowledge-base of logic predicates and relations. Therefore, the FIF “guides” this knowledge-base of rules and inference engine, in such a way that the latter infers the appropriate “conclusions” during transformations of the source programs into scheduled, functionally-equivalent FSMs. To implement these transformations, the back-end compiler consists of definite clauses [2] of the following form:

\[ A_j \leftarrow A_1 \land A_2 \ldots \land A_n \quad (\text{form } 1) \]

where \( \leftarrow \) and \( \land \) are the logical implication and conjunction symbols (\( A \leftarrow B \) means that if \( B \) applies then \( A \) applies) respectively, and \( A_0, \ldots , A_n \) are atomic formulas (Prolog clauses) of the form:

\[ \text{predicate}_\text{symbol}(\text{Var}_1, \text{Var}_2, \ldots , \text{Var}_N) \quad (\text{form } 2) \]

where the positional parameters \( \text{Var}_1, \ldots , \text{Var}_N \) are either variable names (used in the “inference rules”- Prolog clauses), or constants (in the case of the FIF statements-facts) [14].

In essence, FIF consists of a number of such atomic formulas, which are grouped into the FIF tables. Each such table is a list of homogeneous facts which describe a certain aspect of the compiled program. E.g. all prog_stmt facts for a given subprogram are grouped together in the listing of the program operations table. The FIF syntax is formally defined, but its complete description is not the purpose of this paper.

Verifications at different abstraction levels are attempted on the compiled program. Therefore, the FIF “guides” this knowledge-base of rules and inference engine, in such a way that the latter infers the appropriate “conclusions” during transformations of the source programs into scheduled, functionally-equivalent FSMs. To implement these transformations, the back-end compiler consists of definite clauses [2] of the following form:

\[ A_j \leftarrow A_1 \land A_2 \ldots \land A_n \quad (\text{form } 1) \]

where \( \leftarrow \) and \( \land \) are the logical implication and conjunction symbols (\( A \leftarrow B \) means that if \( B \) applies then \( A \) applies) respectively, and \( A_0, \ldots , A_n \) are atomic formulas (Prolog clauses) of the form:

\[ \text{predicate}_\text{symbol}(\text{Var}_1, \text{Var}_2, \ldots , \text{Var}_N) \quad (\text{form } 2) \]

where the positional parameters \( \text{Var}_1, \ldots , \text{Var}_N \) are either variable names (used in the “inference rules”- Prolog clauses), or constants (in the case of the FIF statements-facts) [14].

In essence, FIF consists of a number of such atomic formulas, which are grouped into the FIF tables. Each such table is a list of homogeneous facts which describe a certain aspect of the compiled program. E.g. all prog_stmt facts for a given subprogram are grouped together in the listing of the program operations table. The FIF syntax is formally defined, but its complete description is not the purpose of this paper.

1. start with the initial schedule (inc. the special external port operations)
2. Current PARCS state \( \leftarrow 1 \)
3. Get the 1st state and make it the current state
4. Get the next state
5. If no dependencies then absorb the next state into the current PARCS state; If there are dependencies then absorb the current state into the PARCS state, store the PARCS state, PARCS state \( \leftarrow \) PARCS state + 1; make next state the current state
6. If next state = conditional then call the conditional (true/false branch) processing predicates, else continue
7. If there are more states to process then goto step 4, otherwise finalize the current PARCS state and terminate

Figure 3. Pseudo-code of the PARCS scheduling algorithm

The operations are scheduled using an aggressive algorithm called PARCS (Parallel Abstract Resource-Constrained Scheduling). PARCS will attempt to schedule as many as possible operations in the same control step, satisfying of course the various dependencies and the specific resource (operator) constraints if any, provided by the user. The PARCS scheduling engine which is executed as part of the back-end compiler transformations is briefly described in Figure 3. The inference engine of the back-end compiler consists of a very large number of Prolog predicates with formal logic rules such as the one in form 1. A new design to be synthesized is loaded via its FIF into the back-end compiler’s inference engine. Hence, the FIF’s facts “drive” the logic rules of the back-end compiler which generate provably-correct hardware architectures. The generated technology-independent (VHDL) RTL code is transformed into silicon through the use of commercial RTL HDL synthesis E-CAD tools. For example the following back-end compiler rule:

\[ \text{dont_schedule}(\text{Operation1}, \text{Operation2}) \leftarrow \text{predecessor}(\text{Operation1}, \text{Operation2}). \quad (\text{form } 3) \]

is a “cause-result” logic relation between two predicates. If the “cause” predicate (op1) is predecessor (op2) is valid then it produces (concludes to) the “result” predicate of the rule
dont_schedule (these two operations in the same state). This cause and result/effect relation between these two predicates is characteristic of the way the back-end compiler’s inference engine rules work. By using this type of inference logic the back-end compiler executes provably-correct synthesis transformations on the source programs.

VI. GENERATED ARCHITECTURE AND INTERFACES

The prototype compiler generates RTL hardware descriptions in standard VHDL-93 (IEEE 1076-1993). The generated VHDL code is free from any technology-specific semantics, from any predefined and fixed hardware architecture template, and from any E-CAD vendor tool. Consequently, the generated RTL code is synthesizable with any commercial synthesis tools. The compilation generates structures that are synchronous, Finite State Machine (FSM) – controlled, micro-architectures. The generated architectures communicate with their host computing environment, with each other, and with other external blocks such as main or local memories, using a robust and general handshake mechanism (a “start”-“done” I/O combination), the set of I/O interface ports found in the source code’s subroutine interface, and a set of synchronous or asynchronous memory interfaces with the appropriate communication protocols, all of which are generated automatically and embedded in the FSM’s operations.

The functions of each generated FSM are initiated by the assertion of an input “start” signal, and the completion of the functions is indicated to the host environment by asserting an output “done” signal. The source program’s data resources (e.g. single or multi-dimensional variables) are mapped on local (register) or shared (external) memory blocks. The selection of a mix of the two types of (local and/or shared) memories is done in either (or a combination of) the two following methods:

1. By selecting the coding style of the source code subroutines’ interfaces (e.g. using array parameters)
2. By setting external block (e.g. memory) specific options, which are utilized by the back-end compiler stage

The external port options, given to the back-end compiler via an options file, may contain user parameters, such as memory size, data and address width, synchronous or asynchronous communication type, etc, all of which are automatically generated in the produced HDL model. In this way, the most suitable to the application interface, is generated.

VII. EXPERIMENTAL EVALUATION OF THE METHOD

A large number of applications were run through the compilation flow of Figure 1. The functionality of the generated hardware architectures were verified with simulations, and in all cases, the implemented hardware function matched the function of the source programs. Experiments with three of the benchmarks are discussed here.

<table>
<thead>
<tr>
<th>Module</th>
<th>Initial schedule states</th>
<th>PARCS parallelized states</th>
<th>State reduction rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>c. graphics (with embedded mem)</td>
<td>113</td>
<td>69</td>
<td>39%</td>
</tr>
<tr>
<td>c. graphics (with external mem)</td>
<td>167</td>
<td>115</td>
<td>31%</td>
</tr>
<tr>
<td>nested loops 1st routine</td>
<td>28</td>
<td>20</td>
<td>29%</td>
</tr>
<tr>
<td>nested loops 2nd routine (with embedded mem)</td>
<td>36</td>
<td>26</td>
<td>28%</td>
</tr>
<tr>
<td>nested loops 2nd routine (with external mem)</td>
<td>96</td>
<td>79</td>
<td>18%</td>
</tr>
<tr>
<td>nested loops 3rd routine</td>
<td>15</td>
<td>10</td>
<td>33%</td>
</tr>
<tr>
<td>nested loops 4th routine</td>
<td>18</td>
<td>12</td>
<td>33%</td>
</tr>
<tr>
<td>nested loops 5th routine</td>
<td>17</td>
<td>13</td>
<td>24%</td>
</tr>
<tr>
<td>MPEG 1st routine</td>
<td>88</td>
<td>56</td>
<td>36%</td>
</tr>
<tr>
<td>MPEG 2nd routine</td>
<td>88</td>
<td>56</td>
<td>36%</td>
</tr>
<tr>
<td>MPEG 3rd routine</td>
<td>37</td>
<td>25</td>
<td>32%</td>
</tr>
<tr>
<td>MPEG top routine (with embed. mem)</td>
<td>326</td>
<td>223</td>
<td>32%</td>
</tr>
<tr>
<td>MPEG top routine (with external mem)</td>
<td>462</td>
<td>343</td>
<td>26%</td>
</tr>
</tbody>
</table>

The hardware compilations were executed on a conventional Pentium-4 platform running the MS-Windows-XP-SP2 operating system. The three design benchmarks
include a line-drawing routine from computer graphics area, a set of 5 subroutines utilizing 2-level nested loops and a MPEG video compression application coded in 4 subroutines. An implementation option of the last two benchmarks has been to store the multi-dimensional data arrays in external memory.

The reduction of the number of states (control steps) via the use of the PARCS optimizer is shown in TABLE I, for the routines of the three benchmarks.

<table>
<thead>
<tr>
<th>Design benchmark</th>
<th>Backend (synthesis) compilation time with PARCS optimiser</th>
</tr>
</thead>
<tbody>
<tr>
<td>c. graphics (with embedded mem)</td>
<td>11 secs</td>
</tr>
<tr>
<td>c. graphics (with external mem)</td>
<td>14 secs</td>
</tr>
<tr>
<td>Nested loops with embedded memory</td>
<td>2 secs</td>
</tr>
<tr>
<td>Nested loops with external memory on the 2nd routine</td>
<td>32 secs</td>
</tr>
<tr>
<td>MPEG with embedded memory</td>
<td>3 mins and 16 secs</td>
</tr>
<tr>
<td>MPEG with external memory</td>
<td>9 mins and 24 secs</td>
</tr>
</tbody>
</table>

The durations of the back-end compilations for the three benchmarks, with embedded or external memory options are shown in TABLE II. This demonstrates the ability of the design framework to quickly deliver synthesis results, so that alternative solutions can be explored.

<table>
<thead>
<tr>
<th>Area and speed statistic</th>
<th>MPEG, 4th routine (unoptimised)</th>
<th>MPEG, 4th routine optimized (PARCS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cells</td>
<td>3132</td>
<td>3186</td>
</tr>
<tr>
<td>Combinational area</td>
<td>154345.422880</td>
<td>148026.032483</td>
</tr>
<tr>
<td>Noncombinational area</td>
<td>390809.611002</td>
<td>389015.961706</td>
</tr>
<tr>
<td>Total cell area</td>
<td>545155.033882</td>
<td>537041.994189</td>
</tr>
<tr>
<td>Eq. NAND2x1 gates</td>
<td>~54.515K gates</td>
<td>~53.704K gates</td>
</tr>
<tr>
<td>Comp. time</td>
<td>39 mins</td>
<td>18 mins</td>
</tr>
<tr>
<td>Total cell area</td>
<td>545155.033882</td>
<td>537041.994189</td>
</tr>
<tr>
<td>Minimum clock period</td>
<td>9.19 ns (10 ns constraint with 0.81 ns slack)</td>
<td>9.26 ns (10 ns constraint with 0.74 ns slack)</td>
</tr>
</tbody>
</table>

The back-end compiler is able to produce RTL (V)HDL for both the initial schedule (right before the PARCS optimizer is applied) and the optimized (parallelized) schedule. All of the produced RTL modules were synthesized with the Xilinx XST and Synopsys DC Ultra RTL synthesizers. Due to lack of space in this paper, only the fourth (top-level) routine of the MPEG benchmark Xilinx Synthesis and Place & Route, as well as Synopsys RTL synthesis run statistics are shown here, in TABLES III and IV. The XST and implementation ran via the Xilinx Design Suite 10.1 and was mapped on Xilinx Virtex5 XC5VLX330T, package FF1738, speed -2 device. The Synopsys run via the DC-ULTRA version C-2009.06-SP3 and was mapped on TSMC 1.3 um libraries.

Figure 5 demonstrates in graphical way the reduction of the number of states of the initial schedule after applying the PARCS optimizing engine.

The 2nd routine of the nested-loops benchmark ADA source code is shown in Figure 6. The source code subroutine of the benchmark specification in Figure 6 contains unaltered regular behavioural ADA program code, without additional semantics and compilation directives which are usual in other synthesis tools which compile code in SystemC, HandelC, ParallelC or any other modified program code with additional object class and TLM primitive libraries.

Abstract, algorithmic coding in standard programming languages can be done in a fraction of the time required to model and debug the application directly in cycle-accurate, detailed (scheduled) RTL code, using any hardware description language. This method which automatically transforms whole ADA programs into VHDL RTL, along with the co-processor’s interfaces, is more efficient compared even to coding by very experienced hardware designers, and particularly when the complexity of the hardware machine increases usually over a dozen states. This conclusion is inferred simply by observing that from the very compact program code in Figure 6, the hardware compiler optimizes about 100 initial FSM states and the produced optimized schedule is coded in about 750 lines of
optimized lines of synthesizable VHDL code, something particularly cumbersome to be done manually.

```vhdl
procedure ROUTINE2(ZMAX : IN integer;
                     S, A, K, PM : in out ARRAY_100_X_100;
                     POUT : OUT integer)
is
    P, I, J : integer;
    TEMP1 : integer;
begin
    for I in 2..99 loop
        for J in 2..99 loop
            P := PM(I-1)(J)/2 + PM(I-2)(J-2)/2;
            if TEMP1 < ZMAX then
                PM(I)(J) := 1 - A(I)(J)*P;
            else
                PM(I)(J) := P;
            end if;
        end loop;
    end loop;
    POUT := P;
end ROUTINE2;
```

Figure 6. The 2nd routine of the nested-loops benchmark

VIII. CONCLUSIONS

This paper discussed a method for the automatic and formal generation of custom, massively-parallel hardware architecture implementations, from whole and complete, abstract, high-level programs using compiler-generator and logic programming techniques. This method achieves a formal equivalence of the produced implementations with the executable program specifications, as well as a rapid design and implementation cycle for mixed software & hardware systems. Moreover, this method enables the automatic generation of on-chip, or external memory interface ports along with their addressing and communication protocols.

The generated hardware features a great deal of resource redundancy, meaning that a large number of hardware operators (e.g. adders/subtractors, multipliers, etc.) remain unused during some FSM states. This is because all the used operators are dedicated to certain states. However, the redundancy is compensated with the reduction of the data multiplexing and state control signaling into simple wires. This in turn, achieves very short clock periods during implementation, thus increasing the application’s throughput.

Future developments for this work include adoption of a range of input programming language formats, evaluation of the method for larger application range and size and further improvement of the compiler’s optimization algorithms with more coarse-grained transformations. Moreover, by upgrading of FIF and the compilation options, reusable pre-compiled library components can be used in the design flow. Another enhancement of the current work includes the extension of the input source program semantics to support object-oriented model-based design features. Furthermore, by updating the back-end compiler’s HDL writer, other HDL language formats such as Verilog RTL can be easily generated.

REFERENCES