A Framework for Automated and Optimized ASIP Implementation Supporting Multiple Hardware Description Languages

Oliver Schliebusch,
A. Chattopadhyay, D. Kammler,
G. Ascheid, R. Leupers, H. Meyr
Aachen University of Technology
Integrated Signal Processing Systems
52056 Aachen, Germany
Email: schliebusch@iss.rwth-aachen.de

Tim Kogel
CoWare, Inc.
2121 N. First Street
San Jose, CA 95131

***Abstract*— Architecture Description Languages (ADLs) are widely used to perform design space exploration for Application Specific Instruction Set Processors (ASIPs). While the design space exploration is well supported by numerous tools providing high flexibility and quality, the methodology of automated implementation is limited to simple transformations. Assuming fixed architectural templates, information given in the ADL is directly mapped to a hardware description on Register Transfer Level (RTL). Gate-Level synthesis tools are not able to perform potential optimizations, as the computational complexity grows exponential with the size of the architecture. Information such as exclusiveness, parallelism or boolean relations are spread over multiple modules and therefore hard to determine. In this paper, we present an ASIP synthesis approach from architecture description languages, based on an Intermediate Representation (IR). The IR is the key technology to provide new language-independent high-level optimizations and to realize different hardware description language backends. The feasibility of our approach is proven in a case-study.**

I. Introduction

Application Specific Instruction Set Processors (ASIPs) are increasingly used in complex System On Chip (SoC) designs [1][2]. ASIPs are tailored to particular applications, thereby combining performance and power efficiency of a dedicated hardware solution with the flexibility of a programmable solution [3]. To identify the optimal architecture for a given application, design environments based on Architecture Description Languages (ADLs) are commonly used [4][5][6][7][8]. They enable a fast development of the initial architecture model and a quick incorporation of specification changes. Software tools, such as compiler, assembler, linker and simulator can be generated automatically from ADLs. The tools enable the evaluation of several solutions within the design space in a reasonable amount of time by obtaining valuable figures on the performance of the architecture (HW) and the application (SW). However, since the estimations are measured on a high abstraction level, questions about physical parameters, such as clock speed, area or power consumption cannot be answered accurately.

The commonly accepted entry level for hardware implementation is the Register Transfer Level (RTL). Gate-Level synthesis tools are used to proceed to gate-level and to obtain figures on physical parameters. On RTL, the target architecture is modelled in a Hardware Description Language (HDL). In ASIP design, the HDL code is mostly developed by hand according to the specifications. This increases time-to-market and may lead to consistency problems in case of late specification changes. However, latest approaches try to overcome these drawbacks by automatically mapping the architecture described in an ADL directly to a hardware model in a HDL. Contrary to mature *exploration* tools available in academia and industry, the results achieved by an automated *implementation* of the target architecture, compared to manual implementations by experienced designers, is rather poor. The reason for worse physical parameters is caused by the nature of ADLs. ADLs describe the architecture concerning its functionality and focus on *what* features are included, without specifying *how* they are realized. Moreover, different architectural aspects, such as instruction set, timing, behavior, storage elements, are specified almost independent. Therefore model changes can be realized fast. Mapping independent aspects to a hardware model results in a highly modular description. Each module supports maximum functionality, as the subset of required functionality is not yet known. Actual, the required optimizations are usually performed by state-of-the-art gate-level synthesis tools. Nevertheless the gain achieved is rather small compared to the results expected. The existing module boundaries and the overall size of the architecture prevent possible optimizations concerning the physical parameters. The computational complexity of, for example, exclusiveness- and parallelism-detection is exponentially growing with the size of the architecture. However, the information required is directly available from the ADL model. Therefore, we are proposing an Intermediate Representation (IR), dedicated to unify information given in the ADL and the underlying hardware description to realize new optimization techniques, heading for a dramatic decrease in area consumption. The technology presented in this paper is implemented using the ADL LISA[4][5] and the HDLs VHDL, Verilog and RTL-SystemC.

The paper is organized as follows: Section II discusses related work. The whole synthesis flow is described in section III. The IR definition and its construction are presented in section IV, while sample optimizations can be found in section VI. A short description of the backends is given in section VII. The paper is concluded with a case study (section VIII) and summary in section IX.

II. Related Work

The concept of an Intermediate Representation (IR) is applied numerous times, during the transformation between two levels of abstraction in EDA. For mapping logic-level representation to IC layout, the Milkyway Database [9] is often used. For performing logic-level optimizations, efficient representations like, Binary Decision Diagrams [10] were introduced. Currently, there is no IR for processor synthesis from ADL to RTL hardware description.

There are several ADLs supporting hardware generation. The different ADLs can be organized into those focusing on the architecture, on the instruction-set or a combination of both.

MIMOLA [11] is one example of the languages strongly oriented towards the architecture. As RTL description can be directly plugged into that, usage of an IR is not needed there. Some of the languages strongly oriented towards the instruction-set are ISDL [7] and nML [6]. The synthesis tool HGEN [12] is used to generate synthesizable Verilog code from an ISDL description. The HDL generator GO from Target Compilers Technologies [13], which is an industrial product, is based on the architecture description language nML. The synthesis results are not publicly available. The project Sim-HS [8] is also based on the nML description language and generates synthesizable Verilog models from Sim-nML models. None of the above-mentioned projects approached the introduction of a suitable IR.
Approaches based on an instruction set/architecture combination are as follows. FlexWare [14] is more related to RTL than to the level of ADLs. The PEAS-III [15] and the derived ASIP-Meister [16] work with a set of predefined components. Automatic HDL generation from the ADL EXPRESSION [17] uses functional abstraction of hardware components. Actual instantiation of components is captured by the parameters of the function. This approach is not suitable for multi-layer optimization. There is no information available about supporting several HDL backends.
The novel RTL processor synthesis framework and the IR are based on our previous work in this area, which is published in [18] and [19].

III. RTL Processor Synthesis Framework Using an Intermediate Representation

The proposed synthesis framework is depicted in figure 1. It bases on an IR and separates the flow into three different phases: The first phase covers the *construction* of the IR from an ADL, performed by the frontends. The second phase performs *optimizations* using the IR. Optimization algorithms are implemented once, being independent from both the ADLs as well as the HDLs. Finally, during the third phase of *RTL generation* the IR supports several different HDL back-ends.
A *functional abstraction* of the underlying hardware on RTL is achieved by utilizing processes and signals. Whereas, the *structural abstraction* utilizes entities to describe the architecture structure. In both abstractions, the semantics given in the ADL are not covered by the HDLs available. For example, the instruction set, including opcodes, condition-fields or operand-fields are hardly extractable from RTL. However, this information is absolutely necessary to apply new optimization techniques, as we will show in section VI. The IR is conceived with respect to two major needs:

1) The IR should be a hardware description located at a higher abstraction level compared with HDLs.
2) The semantical meaning about the underlying hardware, extracted from the ADL, has to be preserved in the IR.

By satisfying the first requirement, we preserve the aspects of a hardware description enabling the path to implementation, while the second one integrates the advantages provided by ADLs.

IV. Definition of the Intermediate Representation

As mentioned in section III, the definition of the IR is steered by combining benefits of ADLs and HDLs. The first objective is to define the basic elements used. The second objective is to annotate semantical information to each element. The basic IR elements are depicted in figure 2.

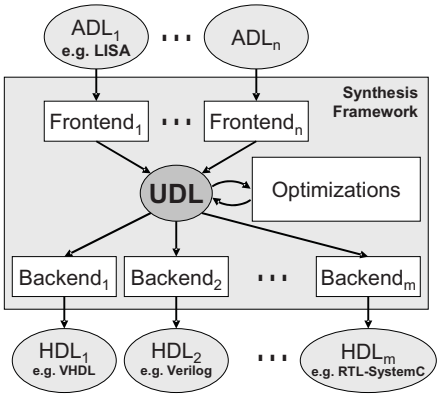

Fig. 1. Synthesis flow based on an IR

In this paper we are using the following notation: A set $\mathcal{X}$ is defined as $\mathcal{X} = \{X_1, X_2, ..., X_{N_X}\}$, where N_X denotes the number of elements in the set. $i, j, k, l, r \,\epsilon\, \mathbb{N}$ and $1 \leq i, j, k, l, r \leq N_X$ denote indices.
The hardware behavior is described on RTL by a set of combinational and sequential *processes* in VHDL or *always-blocks* in Verilog. This leads to the definition:

Definition PROCESS: Each process encapsulates hardware behavior, which is represented by a Control Data Flow Graph (CDFG). $\mathcal{A}$ denotes the set of processes $\mathcal{A} = \{A_1, A_2, ..., A_{N_A}\}$, which is used to model the whole target architecture on RTL.

Information between the processes is exchanged via signals:

Definition SIGNAL: A signal is a connection between processes and dedicated for communication. $\mathcal{S}$ denotes the set of signals required to model the target architecture on RTL, with $\mathcal{S} = \{S_1, S_2, ..., S_{N_S}\}$.

The processes and signals provide a functional abstraction of the architecture on RTL. However, the similarity of process functionality can be used to group them into even more abstract functional blocks, such as decoders, fetch unit or pipeline-controller.
This additional abstraction layer is represented by units:

Definition UNIT: A unit U_i is defined as set of processes $U_i \subset \mathcal{A}$ with $\bigcup_{i=1}^{N_U} U_i = \mathcal{A}$ and $\forall\, U_i, U_j \,\epsilon\, \mathcal{U}$ and $U_i \neq U_j$ is $U_i \cap U_j = \emptyset$. $\mathcal{U}$ denotes the set of all units.

Analogously, information about the purpose of signals can be used to group them as well. Signals required for a particular information exchange are grouped in an IR-path:

Definition IR-PATH: An IR-path P_i is defined as set of signals $P_i \subset \mathcal{S}$ with $\bigcup_{i=1}^{N_P} P_i = \mathcal{S}$ and $\forall\, P_i, P_j \,\epsilon\, \mathcal{P}$ and $P_i \neq P_j$ is $P_i \cap P_j = \emptyset$. $\mathcal{P}$ denotes the set of all paths.

For example, a simple assignment to a register array in LISA such as `R[address]=data;`, is represented in an HDL by three different signals. Those comprise data, address and enable flag indicating the validity of the data and address values. A path groups those signals modelling a particular

transaction.
A ***functional abstraction*** of the architecture is defined by a multigraph:

Definition $\mathcal{G}_{IR}$: $\mathcal{G}_{IR}$ is a directed multigraph $\mathcal{G}_{IR} = (\mathcal{U}, \mathcal{P})$. Here, the vertex set is $\mathcal{U}$ and the edge set is $\mathcal{P}$.

On RTL a hierarchical model structure is provided by nested entities in VHDL)*modules* in Verilog, *sc_module* in RTL-SystemC). Following this principle our IR uses entities to encapsulate the functionality provided by units:

Definition ENTITY: An entity E_i is defined as set of units with $\bigcup_{i=1}^{N_E} E_i = \mathcal{U}$.

Finally, a ***hierarchical abstraction*** of the architecture is defined by a tree:

Definition $\mathcal{T}_{IR}$: $\mathcal{T}_{IR}$ is defined by a directed tree $\mathcal{T}_{IR} = (\mathcal{E}, \mathcal{I})$. Here, the vertex set is $\mathcal{E} = E_1, E_2, ..., E_{N_E}$. $\mathcal{I}$ denotes the set of edges. A directed edge $\langle E_i, E_k \rangle$ represents the relation between E_i and E_k where E_k contains a partial function of E_i. This corresponds to the component instantiation in in VHDL.

Finally, we define the IR as follows:

Definition IR: The IR is defined by a functional abstraction $\mathcal{G}_{IR}$ and hierarchical abstraction $\mathcal{T}_{IR}$. The edges of the multigraph G_{IR} are related to the edges of the tree $\mathcal{T}_{IR}$ as an edge $P_r = \langle U_i, U_k \rangle$ is always bound to a path in $\mathcal{T}_{IR}$, here $I_{l,m} = \langle E_l, E_{l+1}, ..., E_m \rangle$ with $U_i \epsilon\, E_l$ and $U_k \epsilon\, E_m$. This defines a mapping: $P_r \mapsto I_{l,m}$. The hierarchy represented in $\mathcal{T}_{IR}$ leads to $E_i = \bigcup_{E\ is\ child\ of\ E_i} E$.

An example of the IR is depicted in figure 2 including entities, edges I (dotted arrows), units, paths (solid lines) and the processes included in a unit.

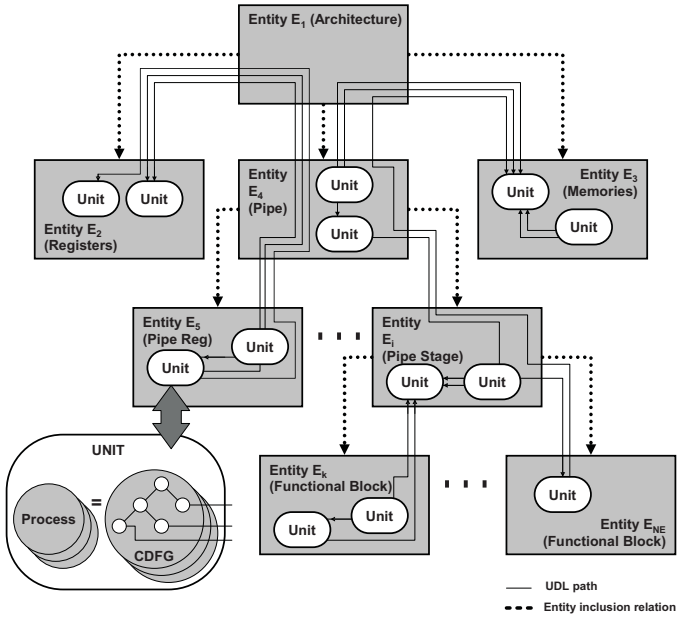

Fig. 2. Example IR of an ASIP

Until now, only the first requirement regarding a higher abstraction level is satisfied. It remains to preserve the semantical information given by the ADL. Thus, we derive specific types from the IR elements entity, unit and path. In other words, a semantical information is annotated to every element.
Table I contains an exhaustive list of currently existing types of IR elements. The types of entities, units and paths have been chosen by implementing and evaluating several architectures existent from our previous work. For example the *pattern_matcher* unit is dedicated to recognize particular instruction bit pattern and to set the *coding* path accordingly. The information provided by the coding path as well as run-time conditions are evaluated by the *decoder* unit to set the correct control signals via the *activation* path. Those signals are, for example, utilized by the *data_path* units. The types of our IR are not fixed and will be extended in future work as needed.

TABLE I
EXHAUSTIVE LIST OF ENTITIES, UNITS AND PATHS

List of IR Elements		
Entities	Units	Paths
testbench	data_path	resource
architecture	register	mem
registers	memory	mem_r
memories	simulation_memory	mem_r_w
pipeline	signal	mem_w
pipe_stage	resource	clock
pipe_register	pin	reset
funct.-block	decoder	activation
	pattern_matcher	pipectrl
	multiplexer	coding
	pipe_controller	ft_pipe_reg
	pipe_register	pipereg_ctrl
	ctrl_stall_flush	pipe_reg_ctrl_flush
	clock	pipe_reg_ctrl_stall
	reset	
	pipe_reg	
	pipe_reg_coding	
	pipe_reg_activation	

V. IR CONSTRUCTION

The ADL-specific frontend constructs the IR from the information given in the ADL. Information directly extracted from the ADL model is called *explicit* information. As mentioned before, ADLs do not cover all the hardware details, as a fast architecture exploration is the primary goal. Thus, missing information must be derived from common knowledge about processor design and adopted to the given architecture description. The information introduced is called *implicit* information. The proposed IR supports the usage of explicit and implicit information in the hierarchical domain $\mathcal{T}_{IR}$ as well as in the functional domain $\mathcal{G}_{IR}$. For example, the LISA frontend uses a hierarchy template which instantiates the `registers` and `memories` entities in the `architecture` entity (see figure 2). This fact allows various optimizations, as different templates may coexist, each for a particular optimization.
Structuring: In the first step, called structuring, the directed tree $\mathcal{T}_{IR}$ is instantiated. The different entity types used in this example are listed in table I, first column. The entity structure

is equivalent to the structure of the final hardware model on RTL generated by the backends.

Mapping: In the second step, the information given in the ADL is mapped onto the functional representation of the hardware $\mathcal{G}_{IR}$. For example, information about resources is used to map the resource-units while information about the instruction set is mapped onto the decoder-units. However, during this phase only a path subset $\mathcal{P}_{Mapping} \subset \mathcal{P}$ is created. The remaining paths $\mathcal{P} \backslash \mathcal{P}_{Mapping}$ are built up in the next phase. The different units and paths currently available are shown in table I, second and third column.

CDFG Creation: Finally, every process must be implemented by a CDFG. If no information about the realization is given in the ADL, a CDFG template must be tailored to the abstract information provided. For example, LISA specifies the instruction set including the binary encoding but does not cover the technique to decode this. Here, CDFG templates must be tailored to the specified instruction set. In general, they have to be compliant to the model description at the ADL level.

If available, the information given in the ADL model is used to realize the process. For example, LISA allows the designer to specify the state update functions of the architecture using the programming language ANSI-C. This behavior description is transformed into a CDFG hardware description.

Additional paths required due to the CDFG instantiation may be created at this point of time. As illustrated in figure 2 (lower left corner), the starting and ending points of an IR-path are hooked into the CDFG.

VI. Optimizations Using the IR

In general, optimizations can be applied to any element in the IR: Entities, Units, Paths or even CDFGs. We categorize optimizations into two groups: *constraint-independent* and *constraint-dependent* optimizations. In general, constraints describe user defined ranges of variables of the applied cost-function. In the context of gate-level synthesis, constraints are typically set to area, clock speed, power-consumption and/or more. Moving to higher abstraction levels in architecture modelling compels the cost-function to be a function of more abstract and also often conflicting variables. Here, constraints can be set to the number of registers instantiated, organization of decoders (single decoder or distributed decoder) and/or placement of the units. We define *constraint-independent optimizations* to improve one or more variables of the cost-function without making any other variable worse. We define *constraint-dependent optimizations* to improve one or more variables of the cost-function by worsening another variable. In the following we describe two different optimizations.

A. Analysis of Resource Scope

It is natural to model complex data-paths in ASIP design. Those data-paths are not only described by combinational processes, but also by sequential ones. The resulting registers with global scope are exclusively used by this data-path. Here, it is beneficial to move those registers from the position with global scope to a local scope, such as the entities covering the data-path. By this, the implementation of the functional unit becomes more compact. Due to the locality of implementation, gate-level synthesis tools are able to perform stronger optimizations (see section VIII).

Considering the traditional ASIP implementation flow starting at RTL, flattening mechanisms during gate-level synthesis can be used to eliminate the boundaries of the model structure. The goal is to detect exclusiveness and dependencies in general. However, for most designs, flattening results in little optimization due to the increased computational effort.

Our approach, based on the IR, enables optimizations across entity boundaries, without the requirement to eliminate these. The analysis of exclusiveness can either be performed using the information provided in the ADL or using the IR elements path and unit. Either way the search space is reduced compared to gate-level synthesis which enables new optimization techniques with higher computational effort. This optimization is constraint-independent.

The scope of *all* resources such as signals, registers, memories can be analyzed. In order to move them as close as possible to the accessing unit the following rule must be applied.

A resource is represented by a particular unit $U_{Res} \epsilon\, E_i$. The resource is accessed by a set of other units $\mathcal{U}_{Access} = \{U_i, U_{i+1}, ..., U_{i+k}\}$. The resource unit U_{Res} can be moved to a particular entity E if $\forall U \epsilon \mathcal{U}_{Access} : U_{Access} \subseteq E$.

If the resource represents a signal and $N_{\mathcal{U}_{Access}} = 1$ then it is even possible to replace the signal by a local communication mechanism within the unit $\mathcal{U}_{Access}$ (e.g. variables). By this, the unit U_{Res} gets deleted as it is not required anymore. This optimization is strongly used in our case study.

The IR enables automatic movement of functionality represented by units within the target architecture. Semantical information is utilized to consider the correct units, to which this optimization can be applied. The encapsulation of processes and signals is also used, as implementation details do not need to be considered during movement. This feature is currently only applied to resource units but may be extended to others in future.

B. Path Sharing

A *process* is defined in section IV and covers a CDFG. Within the scope of a process, a resource is accessed via an IR-path. An optimum number of resource accesses can be determined by using the concept of sharing paths. Two paths can be shared, if they exist in mutually exclusive nodes of the CDFG and if they perform the read/write access to the same resource. The constraint-independent optimization and its algorithm is shown below.

The function `Create_SetOfCongruentPaths()` is called recursively starting from the root node of the CDFG. It builds up a list of sets. Each set contains all the paths, which can be shared. After the sets are built up for mutually exclusive nodes, the sharing is explored between them. Thus, a set can be shared within another set - essentially merging the two sets. Sharing between two sets is investigated using the representative path of that set, which is the first member of the set.

The function `SharePaths()` is called only once for the root node. It takes one set once and shares all the paths of the set with the representative path.

This algorithm improves the overall area of the architecture by reducing the number of interconnects and multiplexers to the resource. Due to multi-level granularity of the proposed IR, this algorithm can be extended to consider the mutual exclusion of *processes*, *units* or even *entities*.

01 // A set M_i contains paths, which can be shared($\cong$):
02 // $M_i = \{P_i, P_{i+1}, ..., P_k\}$,
03 // with $P_l \cong P_{l+1}$, with $i, l, k \epsilon\, \mathbb{N}$ and $i \leq l < k$.
04 // $\mathcal{L}$ and L_{node_i} denote the list of sets
05 // $L_{node_i} = \{M_1, M_2, ..., M_{N_{node_i}}\}$.
06 // L_{node_i} is assigned to a $node_i$.
07 // We choose an element of the set, called
08 // $P_{sharing}$ to replace shared paths.
09

```
10 void Create_SetOfCongruentPaths(L) {
11
12     // recursion termination
13     if node_this is a resource_access {
14         M_current = Create_SingleMemberSet();
15         L.appendSet(M_current);
16         return;
17     }
18     // calling recursively for mutually exclusive child-nodes
19     for each mutually exclusive child-node node_i
20         node_i.Create_SetOfCongruentPaths(L_node_i)
21
22     // take the list of the first child-node, to start sharing.
23     L_Loop = L_node_1;
24
25     for each M_i ϵ L_Loop {
26         for each L_node_l starting from L_node_2 {
27             for each M_k ϵ L_node_l {
28                 if (P_sharing ϵ M_i ≅ P_k_sharing ϵ M_k) {
29                     M_i.appendSet(M_k);
30                     L_node_l.removeFromList(M_k);
31                 }
32             }
33             L_Loop.appendNonSharedSetsOfList(L_node_l);
34         }
35     }
36     L.appendList(L_Loop);
37
38     //include sets of mutually inclusive child-nodes to L
39     for each mutually inclusive child-node node_i
40         node_i.Create_SetOfCongruentPaths(L);
41 }
42
43 void SharePaths(L) {
44     for each M_i ϵ L {
45         for each (P_i ϵ M_i and (P_i ≠ P_sharing)){
46             P_sharing.shares(P_i_node);
47         }
48     }
49 }
```

The above algorithm is explained using the the example in figure 3. In this example, there are four read operations performed for the resource R, depicted in figure 3 using four assignment nodes. Out of these four nodes, nodes n_3 and n_4 are mutually exclusive to each other.

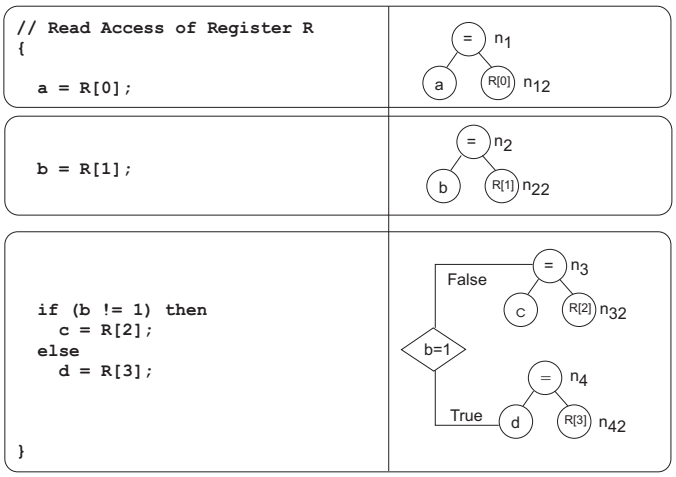

Fig. 3. Path sharing example

The function `Create_SetOfCongruentPaths` is called recursively through each mutually exclusive node. When the function reaches the resource access nodes i.e. node n_{32} or n_{42}, then a single-member set is created with the path to resource R and the recursion terminates (line 13). The set is appended to a global list.

After the sets from mutually exclusive nodes are built up, the creation of congruent sets begin (line 23). Two sets of paths are called congruent, if the first member path of each set is congruent to each other. In this example, two sets are created from mutually exclusive nodes n_3 and n_4. Both of them have one member path reading the same resource, thus being congruent. This results in the two sets getting merged (line 29).

Finally, the function `Create_SetOfCongruentPaths` is again called for mutually inclusive child nodes n_1 and n_2 (line 39). For those, the sharing is not possible in this scope.

The function `SharePaths` takes the list of sets. In this example, there are three lists, two from the two mutually inclusive nodes and one from the merged mutually exclusive nodes. The function starts from the first member path of each set, sharing it with all the other members of the set (line 46). Thus, by applying this algorithm we obtain an optimum number of three paths to the resource R, although four accesses are performed.

VII. HDL GENERATION FROM THE IR

As shown in section IV, the processes and signals required in a HDL are embedded within the basic elements of the IR. To extract the hardware description of the target architecture, the cover of units and IR-paths must be dropped to expose processes and signals. This is the first time in the RTL processor synthesis flow, that the semantical information is omitted.

An Entity E_i is mapped to an *entity* in VHDL, a *module* in Verilog or *sc_module* in RTL-SystemC. The processes, previously covered by a Unit U_i, are mapped to *processes* in VHDL, *always blocks* in Verilog or *sc_methods* in RTL-SystemC. The signals, grouped in an IR-path P_i, generates *signals* and *ports* in VHDL, *wires* and *regs* in Verilog or *sc_signals* and *sc_ports* in RTL-SystemC.

When generating a specific HDL, the language-specific requirements of various HDLs need to be considered. One of the prime constraints is the type-propagation. The HDLs Verilog and RTL-SystemC are loosely typed, whereas in VHDL, the data type is imposed strongly. To cope with this constraint, a specific type-propagation function is inserted before generating a particular HDL.

In addition to the RTL description in various languages, the scripts for driving the RTL simulation and gate-level synthesis are generated automatically for VHDL, Verilog and SystemC. Due to the proper definition of the IR and the backend interfaces, supporting the automatic generation of additional HDLs is seamless.

VIII. CASE STUDY

The example architecture is derived from the Motorola M68HC11 architecture [20]. Our goal was to reuse legacy application code for a bluetooth application, while increasing the performance of the solution. Thus, we developed an architecture compatible on assembly level. The existent application and compiler were reused, whereas the assembler, linker and the new hardware were generated automatically from the LISA model.

While developing the architecture, state-of-the-art architectural features and modern design aspects have been incorporated into the ASIP. The implementation is completely different

TABLE II
M68HC11 RTL MODEL SIZE AND GATE-LEVEL SYNTHESIS RESULTS WITHOUT OPTIMIZATIONS

HDL	RTL model line count	gate-level timing	gate-level gate-count
LISA	7177	-	-
VHDL	47755	5.59 ns	24870
Verilog	55098	5.42 ns	25364
RTL-SystemC	45098	5.21 ns	25830

from the original M68HC11 architecture. The architecture is pipelined with three pipeline stages *fetch*, *decode* and *execute*. The instruction set contains 16/32 bit instructions and is compatible to the original instruction set on assembly level. We reorganized the coding of the architecture to achieve a higher instruction throughput compared to the original architecture. Also, the bus bit-width was increased from 8 bit to 16 bit. The fetch unit is responsible for reorganizing the 16 bit bundles to 32 bit instructions decoded in the second stage. The speedup achieved by this implementation was around 62%.
The M68HC11 compatible architecture (LISA model line count: 7177 lines) was generated completely in VHDL, Verilog and RTL-SystemC without performing any manual optimizations. The RTL model size and the gate-level results achieved are shown in table II. As the results are equal within the precision of the Synopsys DesignCompiler[21], we focus on VHDL in the following paragraphs.

TABLE III
VHDL M68HC11 RTL MODEL SIZE AND GATE-LEVEL SYNTHESIS RESULTS

optimizations	RTL model line count	gate-level timing	gate-level gate-count	
none	47755	5.59 ns	24870	0.0%
+ scope analysis	36384	5.69 ns	24344	-2.1%
+ path sharing	33645	5.74 ns	22343	-10.2%

Two different optimizations have been applied. First we enabled the scope analysis, which reduced the amount of area by 2.11%. Second, we shared the paths to resources wherever possible. This resulted in a further decrease by 2437 gates. The results are shown in table III. Although we only applied basic optimizations, we already automatically decreased the area by 11.9%, without any changes to the LISA model.
Realizing the target architecture in three different HDLs and including optimizations, would not be realizable without a proper IR and synthesis framework. In order to find the best solution for our application we iterated over six models within two weeks including RTL processor synthesis and gate-level synthesis in every iteration. This design efficiency, the better synthesis results and the possibility for further optimizations are the basis for a paradigm shift in ASIP implementation.

IX. SUMMARY AND FUTURE WORK

The RTL processor synthesis framework based on an IR was developed to apply various optimizations to ADL based ASIP design. In this paper we present a novel framework, IR and case-study proving the outstanding flexibility of our approach. Modern processor design incorporates a lot of architectural features driven by market requirements. For example, a JTAG interface and debug mechanism are mandatory for new designs to cope with the increasing complexity. The proposed IR may not only be used for optimizations, but also to integrate such features into the automated implementation process. We already realized the automatic generation of a JTAG interface and debug mechanism into our approach, which is beyond the scope of this paper. The new IR based ASIP synthesis framework is embedded into the LISATek product family of CoWare Inc.[22] .

REFERENCES

[1] Michael J. Bass and Clayton M. Christinsen, "The future of the microprocessor business," in *IEEE Spectrum*, April 2002, pp. 34–39.
[2] P. Paulin, M. Cornero, and C. Liem, "Trends in Embedded System Technology: An Industrial Perspective," in *Hardware/Software Co-Design*, M. G. M. Sami, Ed. Kluwer Academic Publishers, 1996.
[3] Arthur Abnous and Jan Rabaey, "Ultra-Low-Power Domain-Specific Multimedia Processors. Proceedings of the VLSI Signal Processing Workshop." in *IEEE Spectrum*, Oct. 1996, p. 461470.
[4] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, A. Wieferink, and H. Meyr, "A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language," *IEEE Transaction on Computer-Aided Design*, vol. 20, no. 11, pp. 1338–1354, Nov. 2001.
[5] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.
[6] A. Fauth and J. Van Praet and M. Freericks, "Describing Instruction Set Processors Using nML," in *Proc. of the European Design and Test Conference (ED&TC)*, Mar. 1995.
[7] G. Hadjiyiannis and S. Hanono and S. Devadas, "ISDL: An Instruction Set Description Language for Retargetability," in *Proc. of the Design Automation Conference (DAC)*, Jun. 1997.
[8] V. Rajesh and R. Moona, "Processor Modeling for Hardware Software Codesign," in *Int. Conf. on VLSI Design*, Jan. 1999.
[9] *Milkyway: www.synopsys.com*, Synopsys.
[10] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, June. 1978.
[11] R. Leupers and P. Marwedel, "Retargetable Code Generation based on Structural Processor Descriptions," in *Design Automation for Embedded Systems*. Kluwer Academic Publishers, Jan. 1998, vol. 3, no. 1, no. 1.
[12] A. Fauth and M. Freericks and A. Knoll, "Generation of Hardware machine Models from Instruction Set Descriptions," in *IEEE Workshop on VLSI Signal Processing*, 1993.
[13] *www.retarget.com*, Target Compiler Technologies.
[14] P. Paulin, C. Liem, T. May, and S. Sutarwala, "FlexWare: A Flexible Firmware Development Environment for Embedded Systems," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goosens, Eds. Kluwer Academic Publishers, 1995.
[15] A. Kitajima and M. Itoh and J. Sato and A. Shiomi and Y. Takeuchi and M. Imai, "Effectiveness of the ASIP Design System PEAS-III in Design of Pipelined Processors," in *Proc. of the Asia South Pacific Design Automation Conference (ASPDAC)*, Jan. 2001.
[16] "ASIP Meister," www.eda-meister.org.
[17] P. Mishra, A. Kejariwal, and N. Dutt, "Rapid Exploration of Pipelined Processors through Automatic Generation of Synthesizable RTL Models," in *International Workshop on Rapid System Prototyping*, June. 2003.
[18] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr, "Architecture Implementation Using the Machine Description Language LISA," in *Proc. of the ASPDAC/VLSI Design - Bangalore, India*, Jan. 2002.
[19] O. Schliebusch, A. Chattopadhyay, M. Steinert, G. Braun, A. Nohl, R. Leupers, G. Ascheid, and H. Meyr, "RTL Processor Synthesis for Architecture Exploration and Implementation," in *Proc. of the Conference on Design, Automation & Test in Europe (DATE) - Designers Forum*, Feb. 2004.
[20] *Microcontroller 68HC11: www.motorola.com*, Motorola.
[21] *www.synopsys.com*, Synopsys.
[22] *www.coware.com*, CoWare Inc.