Optimization Techniques for ADL-driven RTL Processor Synthesis

Oliver Schliebusch, Anupam Chattopadhyay, Ernst Martin Witte, David Kammler
Gerd Ascheid, Rainer Leupers, Heinrich Meyr
Institute for Integrated Signal Processing Systems
RWTH Aachen University
52056 Aachen, Germany
schliebusch@iss.rwth-aachen.de

Abstract

Nowadays, Architecture Description Languages (ADLs) are getting popular to speed up the development of complex SoC design, by performing the design space exploration in a higher level of abstraction. This increase in the abstraction level traditionally comes at the cost of low performance of the final Application Specific Instruction-set Processor (ASIP) implementation, which is generated automatically from the ADL. There is a pressing need of novel optimization techniques for high level synthesis from ADLs, to compensate for this loss of performance. Two important aspects of these optimizations are the efficient usage of available structural information in the high level architecture descriptions and prudent pruning of overhead, introduced by mapping from ADL to Register Transfer Level (RTL). In this paper, we present two high level optimization techniques, path sharing and decision minimization. These optimization techniques are shown to be of lower complexity, by at least two orders, compared to similar optimization during gate-level synthesis. The optimizations are tested for a RISC architecture, a VLIW architecture and two industrial embedded processors, Motorola M68HC11 and Infineon ICORE. The results indicate a significant improvement in overall performance.

1 Introduction

The growing system complexity coupled with application-specific requirements have greatly increased the development costs and reduced the time-to-market for Application Specific Instruction-set Processors (ASIPs), an important building block of System-on-Chip (SoC). Electronic System Level (ESL) tools are used more and more in order to meet this serious challenge of design complexity [1]. It is important to take the advantage of shrinking fabrication process and at the same time, meet tight development cycle constraints. ESL tools based on Architecture Description Languages (ADLs) offer one promising approach. ADLs [2] [3] [4] [5] are employed to model the ASIP in a higher level of abstraction - thereby allowing a virtual reduction of system complexity. The ESL tools, which are based on ADLs, allow automatic generation of the software tool-suite e.g. compiler, assembler, linker, simulator alongwith the automatic generation of the RTL description of the targeted ASIP. There is another important category in ASIP design approach, where a basic processor core is tuned to suit the application [6] [7]. The first approach holds advantage over the second one by covering the complete design space and reaching an optimal solution. The latter approach restricts the designer within a pre-defined design space, thereby achieving faster design convergence. In this paper, we focus on the the ADL-based approach. *RTL processor synthesis* refers to automatic generation of RTL from an ADL.

The approaches based on ADLs, while allowing fast design space exploration due to the high level of abstraction, often compromise or neglect the quality of final ASIP implementation. Although the usage of ADLs during design space exploration is dominant, yet there are very few commercial endeavors to adopt an automatically generated RTL description. The prevalent ADLs are presented in the following paragraph. Here we limit our discussion among the prominent and contemporary ADLs, which does support automatic generation of RTL description.

The RTL generator GO, from Target Compilers Technologies [8], is based on the language nML [4]. An nML description, apart from static storage elements, contains *action* attributes, which refer to the data path of the processor. No publication is available, which discusses the optimization approaches during RTL generation from nML. Sim-nML [9], which is an extended version of nML, allows the designer to access predetermined functional units within the behavior of an instruction. While this enables an explicit resource sharing option in the language, it limits the designer to a fixed functional unit specification and also overlooks the possible options of fine-grained optimizations. The synthesis tool HGEN [10] generates Verilog code from an ISDL description. ADL-based optimization is not covered by the HGEN approach. EXPRESSION is an ADL, based on a combination of instruction-set and architecture description. The RTL generation from EXPRESSION [3] do not address any optimization framework [11]. ASIP-Meister [12], a synthesis framework for rapid prototyping of the processor, allows stepwise selection of the processor architecture details, leading to a synthesizable HDL code [13]. Any usage of the high level architectural details to optimize the HDL description is not publicly available. In general, for all the abovementioned approaches, the usage of automatically generated RTL description in the final ASIP implementation is rare.

The limited use of an automatically generated RTL description as final implementation is mainly due to the poor quality of the generated RTL compared to the manually written code. The quality of RTL also reflects in a suboptimal gate-level net-list, since the architectural information is not visible in the RTL description. There is a pressing need for high-level optimizations during ADL-driven RTL processor synthesis. The consequences are two-fold. First of all, during rapid exploration phases it is rewarding to take the accurate performance figures e.g. timing, power, area from the automatically generated RTL description. High-level optimizations allow these results to be more precise. Secondly, effective ADL-based optimization promotes the usage of automatically generated RTL description, which is required to adapt with the growing system complexity. This served as our motivation to develop a well-defined optimization framework. We chose LISA [14] as the ADL for automatic generation of *optimized RTL*. The optimizations, which we perform during RTL processor synthesis, can be categorized as *structural* and *behavioral*.

The structural optimization exploits the explicit exclusiveness among different parts of a single instruction and exclusiveness among different instructions of the processor, as will be shown.

The behavioral optimization removes the overhead inherent in an abstract software-like description, while mapping to an RTL one. This basically involves optimizations, similar to those performed by a compiler, e.g. constant folding, constant propagation, if-simplification etc. In this paper, we present a behavioral optimization algorithm, which is highly effective for RTL synthesis.

The contribution of this paper is to present an optimization framework and two optimization algorithms for high-level synthesis from ADL.

The paper is organized as follows: section 2 briefly introduces the features of LISA, which are necessary for understanding the optimization framework. Section 3 describes the optimization framework and analyzes the complexity of optimizations performed during RTL processor synthesis compared to gate-level synthesis. Sections 4 and 5 explain the algorithms, we developed for the optimization. The results are presented and analyzed in section 6. This paper ends with summary and outlook.

2 Brief overview of LISA

In this section, a brief overview of the architecture description language LISA is provided. The language elements, which are relevant for this paper are only covered.

2.1 LISA Operation Graph

In LISA, an *operation* is the central element to describe the timing and the behavior of a processor instruction. The instruction may be split among several LISA operations. The resources (registers, memories, pins etc.) are declared globally in the *resource section*, which can be accessed from any LISA operation.

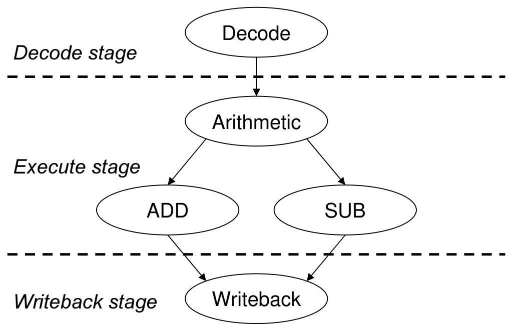

Figure 1. LISA operation DAG

The LISA description is based on the principle that a specific common behavior or common instruction encoding is described in a single operation whereas the specialized behavior or encoding is implemented in its *child* operations. With this principle, LISA operations are basically organized as an n-ary tree. However, specialized operations may be referred to by more than one *parent* operation. The complete structure is a Directed Acyclic Graph (DAG) $\mathcal{D} = \langle V, E \rangle$. V represents the set of LISA operations, *E* the graph edges as set of child-parent relations. These relations represent either *Behavior Calls* or *Activations*, which refer to the execution of another LISA operation. For a LISA operation P, the set of children $\mathcal{C}_P$ can be defined by $\mathcal{C}_P = \{c \mid c \in V \wedge (P, c) \in E\}$.

Figure 1 gives an example of a LISA operation DAG. As shown, the operations can be distributed over several pipeline stages.

2.2 Instruction Coding Description

The instruction encoding of a LISA operation is described as a sequence of several *coding fields*. Each coding field is either a terminal bit sequence with “0”, “1”, “don’t care”(X) bits or a nonterminal bit sequence referring to the coding field of another child LISA operation.

An example of a coding tree is given in the figure 2. In this example, the *Add* and *Sub* operations have only terminal codings whereas *Load*, *Arithmetic*, *Logical* and *Control* consist of both terminal and nonterminal coding fields.

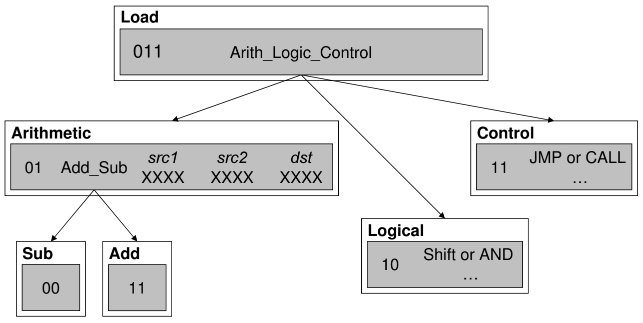

Figure 2. LISA coding tree

2.3 Activations and Behavior Calls

A LISA operation can *call*/*activate* other operations in the same or a latter pipeline stage. In either case, the child operation may be called/activated *directly*, via a *group* or via *compile-time conditional statements*. A *group* collects several LISA operations, with the elements being mutually exclusive. The elements are distinguished by a distinct binary coding, thus forming a coding tree. A *compile-time conditional statement* contains a condition, which can be evaluated during the compilation of the LISA description.

2.4 Behavior Description

The behavior description of a LISA operation corresponds to the datapath of the ASIP. The behavior description is a non-formalized element of LISA language (contrary to formalized elements like *coding*, *activation* etc.), where plain C code can be used. Resources such as registers, memories, signals and pins as well as coding elements can be accessed in the same way as ordinary variables.

3 Optimization Framework

The optimizations performed during RTL processor synthesis from LISA are designed to exploit as much information as possible from the ADL description. The optimizations are classified into two broad categories **- structural** and **behavioral**. The structural optimizations benefit directly from the LISA operation hierarchy. Hence, it is crucial to store and retrieve the structural information in an efficient way. We show that exploiting the same structural information during gate-level synthesis is much more complex than RTL synthesis from an ADL.

A key structural information in LISA is the *mutual exclusiveness* of LISA operations. We represent the exclusiveness information in the form of a **global conflict graph**, where a conflict edge between two operations indicates that those are not mutually exclusive.

$$A \odot B \cong \text{Operation A conflicts with Operation B} \tag{1}$$

$$\begin{aligned} \mathcal{G}_{conflict} &= \langle V, E_{conflict} \rangle \\ E_{conflict} &= \{(x,y) | x, y \in V, x \odot y\} \end{aligned} \tag{2}$$

3.1 Exclusiveness Analysis

In LISA, exclusiveness information can be obtained explicitly from the structural model description and implicitly from data flow.

Explicit information on exclusiveness relations is provided by dedicated elements of the structural LISA model description. There are several sources for this information:
Instruction Set Encoding: The coding tree is a decision tree. For each decision, *only one* single child out of multiple children (a LISA group) is selected, thus providing explicit exclusiveness information.
Conditional Behavior/Activation: Compile-time conditional statements such as *IF/ELSE* or *SWITCH/CASE* may enclose activations or behavior calls, so that the exclusiveness among activated LISA operations is obvious.

As shown in figure 3, the operation *Load* contains a coding field *Arith_Logic_Control* referring to three different child operations. These three different child operations have different coding patterns and therefore, are mutually exclusive.

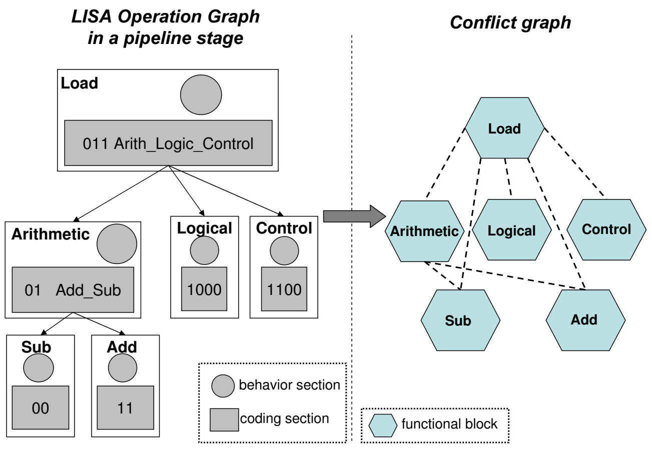

Figure 3. Explicit exclusiveness extraction

Implicit exclusiveness originates from dependencies in distinct run-time and compile-time conditional statements (within the LISA behavior description). Usually it is highly complex to detect such relations.

General Conditional Statements: Depending on the exact conditional expression, two conditional blocks may be mutually exclusive. This information can only be obtained by truth table comparison for the conditions. The conditions can possibly be linked to already obtained explicit exclusiveness information, thus reducing the complexity of boolean comparison. This implicit exclusiveness information is useful for the behavioral optimizations. As shown in figure 4, the operation *Load* contains a C-based behavior, which can be further separated in several blocks. The exclusiveness of the blocks is determined in this case, which is propagated within the block to ascertain the exclusiveness among the statements.

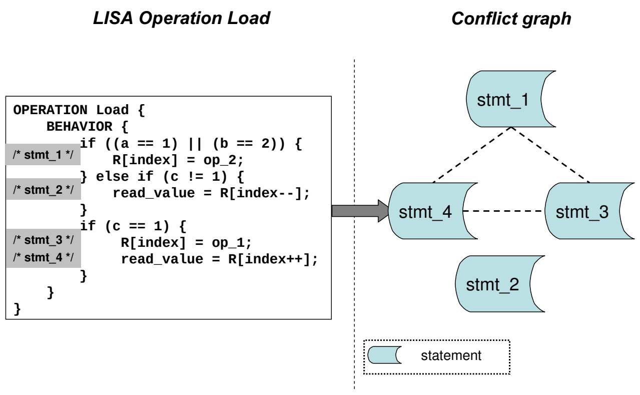
```
OPERATION Load {
    BEHAVIOR {
        if ((a == 1) || (b == 2)) {
/* stmt_1 */    R[index] = op_2;
        } else if (c != 1) {
/* stmt_2 */    read_value = R[index--];
        }
        if (c == 1) {
/* stmt_3 */     R[index] = op_1;
/* stmt_4 */     read_value = R[index++];
        }
    }
}
```

Figure 4. Implicit exclusiveness extraction

The optimization framework heavily exploits the structural information obtained from the ADL. The structural exclusiveness information is represented in the form of a global conflict graph of the LISA operations. Within each operation, another layer of exclusivenss information is maintained on the basis of implicit exclusiveness. Therefore, it is possible to efficiently determine the exclusiveness, even across the LISA operations.

3.2 Comparison of the Order of Complexity

Here, we show that the order of complexity for extracting the global exclusiveness, is lower in case of ADL by at least two orders, compared to RTL. For comparing the order of complexity, we first convert the LISA DAG into an n-ary LISA operation tree by introducing multiple instances of common child operations. Including the multiple instances, let us assume the total number of LISA operations to be n_{op}. For LISA, decisions for exclusiveness of operations OP_A and OP_B are taken in their common parent OP_X without any computation, as exclusiveness is given inherently by the structure. Now, the conflict graph with exclusiveness information is created for each LISA operation only once and merged at the parent operation. Hence, the complexity for the global conflict graph creation is entirely dependent on the structure of the LISA operation tree. For worst case analysis, we take a degenerated binary tree, where the left son of each node is a leaf of the tree. The level of this tree is $\lfloor n_{op}/2 \rfloor$. The recursion step on level L has to work on $L+1$ child operations. Thus, the worst-case complexity of exclusiveness detection from the ADL is as shown in equation 3.

$$\sum_{L=0}^{n_{op}/2} (L+1) \Rightarrow \mathcal{O}(n_{op}^2) \tag{3}$$

On the RTL, LISA operations are mapped to VHDL processes. Thus, the instruction is decoded via several processes, with each process maintaining its own behavior. This implicitly represents the LISA coding tree. However, the link given by the common parent process P_X is not available. The information, whether process P_A and P_B are exclusive, is given by separate boolean decoder signals, e.g. $A_{decoded}$ and $B_{decoded}$. These signals are boolean functions of the instruction coding. The processes P_A and P_B are exclusive if the conjunction of their boolean decoded signals (active high) is zero.

$$P_A \odot P_B \Leftrightarrow A_{decoded} \wedge B_{decoded} \equiv 0 \tag{4}$$

The boolean functions for the decoder signals require at least $b_A = \log_2(n_{children,A})$ and $b_B = \log_2(n_{children,B})$ binary decisions, where $n_{children,i}$ denotes the number of children of operation OP_i. In the worst case, the boolean functions for both decoder signals cannot be minimized at all. When both functions are combined in a conjunction for exclusiveness detection (equation 4), $2^{b_A} \times 2^{b_B}$ computations are performed. Therefore, a single check for exclusiveness of $A_{decoded}$ and $B_{decoded}$ requires $n_{children,A} \times n_{children,B}$ computations.

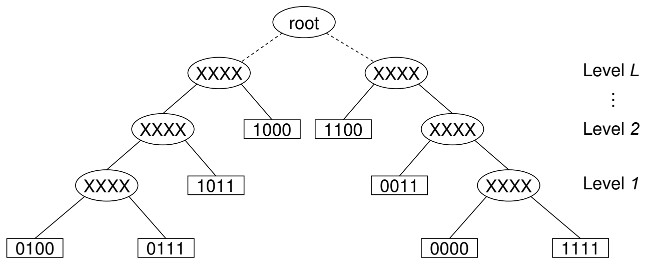

Figure 5. Example of a coding tree

The total exclusiveness detection problem on RTL is most complex for a coding tree containing non-minimizable boolean functions for all decoder signals. Such a case is shown in figure 5.

Obviously, the root of the coding tree is not considered for the exclusiveness detection. Leaving the coding root, the number of nodes with nonterminal codings is equal to the level L, which is $L = \lfloor n_{op}/4 \rfloor$. For each nonterminal coding element, the decoder signal is essentially a non-minimizable boolean function with $level + 1$ terms. The number of term comparisons for all pairs of nonterminal coding elements from different lists is given by the iteration over all levels i and j as shown in equation 5.

$$\sum_{i=1}^{L}\sum_{j=1}^{L}(i+1)\cdot(j+1) = \frac{(L+2)^2\cdot(L+1)^2}{4} \qquad (5)$$

Hence, the worst-case complexity for exclusiveness detection at RTL is given by equation 6.

$$\mathcal{O}(L^4) \Rightarrow \mathcal{O}(n_{op}^4) \qquad (6)$$

Comparing equation 3 and 6, it is clear that exclusiveness analysis within ADL descriptions has a vital performance advantage. This difference in order of complexity is arising from the fact that, for ADL, the global conflict graph is built in a bottom-up fashion by considering each LISA operation only once. Whereas, for RTL, this dependency between processes is not explicit and to create a global conflict graph, the exclusiveness among each pair of VHDL processes has to be checked separately.

Note that, the comparison of complexity, which is deduced above, assumes a certain structure of the generated RTL description. A more general comparison will require considering different *RTL description style* of ASIPs, which is neither standardized nor universal. Additionally, the quality of the hand-written RTL description strongly depends on expertise of the designer. To generally establish the above comparison of complexity, we undertook the task of comparing the automatically generated RTL description to that of commercial hand-written description. Our comparative results with hand-written RTL description establishes the fact that it is more complex for gate-level synthesis tools to exploit and use the same information as performed by an automatic RTL Processor Synthesis tool.

4 Structural Optimization: Path Sharing

The LISA behavior description allows access to two different kinds of data elements.The first kind is a local variable declared in the behavior section. Secondly, the global resources declared in LISA (registers, memories, pins etc.) are accessible from a behavior description. The LISA behavior section is eventually mapped to a VHDL *process*. An access to a global resource from LISA behavior section is mapped internally to a $path$. The protocol for each $path$ is implemented with three VHDL *signals*: *Read/Write Enable Signal*, *Address Signal (for array resources)* and *Data Signal*.

The different accesses to a single resource from different LISA operations are denoted by the paths. Hence, the number of paths strongly correlates with the final size of the multiplexer associated with the particular resource. The multiplexer size can be reduced using the structural information present in the global conflict graph as well as in the implicit exclusiveness.

Path sharing is performed in two steps as following.

Local Path Sharing: The paths originating from a single LISA operation are shared by utilizing the *implicit exclusiveness* information. Here, the sharing does not require introduction of any additional multiplexer. Rather, the same path is used for mutually exclusive global resource accesses.

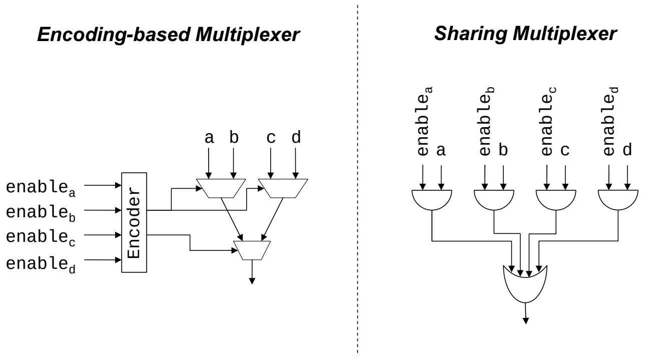

Figure 6. Sharing multiplexer implementation

Global Path Sharing: Global sharing offers the opportunity to share the paths among mutually exclusive LISA operations. Utilizing the LISA operation conflict graph, the minimum number of path accesses is determined by applying a graph coloring algorithm. For small number of nodes, *Brown's Backtracking* [15] and for large ($\geq$ 30) number of nodes, *Recursive Largest First* [16] algorithm is applied. All the paths of same color can be shared. Because of our knowledge about the exclusiveness of the path accesses, a special *sharing multiplexer* can be implemented instead of encoding-based multiplexer. As shown in figure 6, this sharing multiplexer finally maps to a 2-level logical structure in RTL. The first level does the *logical AND* operation between every enable signal with the corresponding data signal. The second level performs *logical OR* operation among the outputs of the first level. For array resources, a similar sharing multiplexer is implemented for the address signal.

5 Behavioral optimization: Decision Minimization

The LISA behavior description consists of plain C-code. It may contain accesses to the global resources inside conditional blocks. A straightforward mapping of a *conditional* resource access to a VHDL description will result in conditional assignment to data, address and enable signal, whereas it will suffice to conditionally assign only the enable signal. This is visible from the example in figure 7. As also depicted in figure 7, this redundancy can be eliminated to limit a *decision making* to absolutely necessary signals.

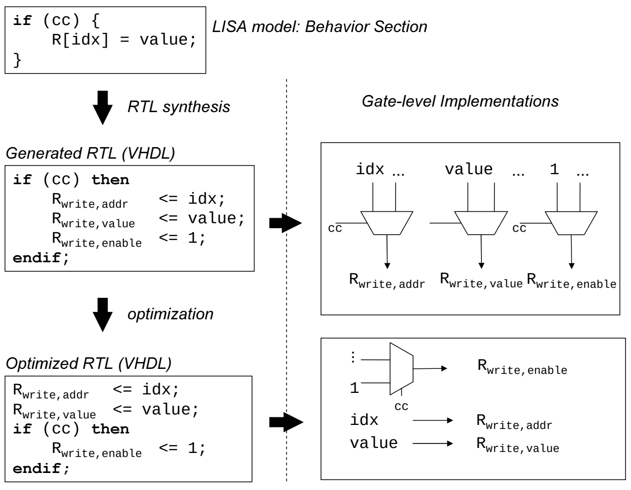

Figure 7. Decision minimization

The aforementioned idea of decision minimization is manifestation of compiler-based optimizations, which perform loop-invariant code motion. Understandably, this has been exploited extensively in high level synthesis frameworks [17] [18]. The notable difference with the earlier works is that, we perform the decision minimization on an intermediate representation with more precise knowledge of definite hardware signals.

The optimization algorithm is a recursive traversal through nested conditional blocks present in the LISA behavior section. Moving conditional assignment of signals out of the associated conditional block is possible only if all input dependencies for the assignment are resolved outside the condition. Since variables introduce such dependencies, first variable assignments need to be moved out of conditional statements, if possible.

For more than one statement moving out of the conditional block, the order of statements is preserved. The algorithm starts with the innermost block and runs till there is no more movable statement.

6 Results

The optimizations discussed in this paper are tested with four different architectures, as described below.

LTRISC is a 32-bit 4-stage pipelined RISC processor with basic support for arithmetic, load-store and branch operations. LTVLIW is a 4-stage 16-bit Harvard VLIW architecture with an array of sixteen 16-bit registers. The instruction set defines two types of concurrent operations. The first type encapsulates arithmetic operations, the second one groups load and store operations. M68HC11 [19] is a well-known micro-controller. An assembly-level compatible 16-bit architecture of M68HC11 is developed for this experiment. The ICORE [20] architecture is dedicated for Terrestrial Digital Video Broadcast (DVB-T) decoding. It is based on a Harvard architecture with a 4-stage pipeline implementing a set of general purpose arithmetic instructions as well as specialized trigonometric operations.

We automatically generated the complete RTL description, using the CoWare/LISATek RTL synthesis framework [21], of the above architectures and applied the proposed optimizations. The RTL description is synthesized with the Synopsys Design Compiler [22] and mapped to a technology library, which is based on a 0.18 μ process. All syntheses are run under worst case conditions. The results for LTRISC and LTVLIW are presented in table 1 and in table 2.

Architecture	Area (K Gates)			
	No Opt.	Path Sharing	Decision Minimization	Both Opt.
LTRISC	26.32	23.89	23.64	21.51
LTVLIW	9.09	8.07	7.59	7.78

Table 1. Optimization effects on area

The results indicate that the optimizations have a strong effect on the architectures of different domains. In general, the path sharing optimization is more effective if the architecture has a balanced instruction coding tree. The amount of optimization achievable in decision minimization depends on the complexity and level of nesting present in the LISA behavior descriptions. For both optimizations applied together, the path sharing is performed first by exploiting the exclusiveness information and then the decision minimization is done. By this, the decision minimization can achieve more benefit because, already shared paths in exclusive blocks result in further condition-independent statements.

Architecture	Clock Period (ns)			
	No Opt.	Path Sharing	Decision Minimization	Both Opt.
LTRISC	6.03	6.02	6.10	6.20
LTVLIW	4.02	4.25	4.04	4.12

Table 2. Optimization effects on clock period

As can be observed from table 2, we have at least as good results w.r.t. timing as compared to RTL synthesis without enabling the proposed optimizations.

A study to test the quality of the generated RTL code compared to the manually written code is performed for the M68HC11 and the ICORE. The M68HC11 is originally an 8-bit micro-controller, available as a DesignWare component [23]. The DesignWare component runs at clock frequencies up to 200 MHz (at 0.13 μm). We compared the speed-up of automatically generated M68HC11 architecture to that of hand-written M68HC11 (table 3) using GNU M68HC11 simulator [24]. For a *spanning tree* application, the automatically generated description ran for much less number of cycles to perform the same task compared to original architecture, therefore being **2.41** times faster overall.

Architecture Version	Cycle-count	Speed (MHz)
M68HC11 (GNU simulator)	522608	200
ISS-68HC11 (with optim.)	195401	180

Table 3. M68HC11 runtime comparison

For comparing the area, we referred to the M68HC11 DesignWare component. The core CPU occupies between 15K and 30K gates depending on the speed, configuration and target technology. On the other side, the RTL description of M68HC11 is automatically generated from LISA with and without enabling the optimizations. The results are summarized in table 4.

Architecture Version	Area (K Gates)	Clock Period (ns)
ISS-68HC11 (unopt.)	24.52	5.82
ISS-68HC11 (opt.)	19.55	5.57
Original M68HC11	15.00	5.00
ICORE (unopt.)	50.85	6.07
ICORE (opt.)	39.40	6.08
ICORE hand-written	42.00	8.00

Table 4. Performance comparison

Table 4 also shows the area and timing delay for three different versions of the RTL description of the ICORE. The ICORE instruction-set architecture is first described using LISA. The first version represents automatically generated RTL code from LISA. The second version is the RTL code, which is automatically generated with the proposed optimizations enabled. The third version has been completely written manually in VHDL. The synthesis results show that the automatically generated RTL description is even smaller than the hand-written one. An ADL-based structural optimization performs important optimizations by sharing *path*s across the boundaries of functional blocks. Performing the same optimization manually would be complex and error-prone.

7 Conclusion

In this paper, we presented two optimization schemes for RTL processor synthesis from ADL LISA. The optimization techniques are shown to be more efficient during ADL-driven RTL processor synthesis than similar optimizations during gate-level synthesis. The efficacy of these optimizations is tested over a RISC architecture, a VLIW architecture and two industrial embedded processors, Motorola M68HC11 and Infineon ICORE. The results indicate a significant area improvement without any loss in speed. The comparison with hand-written RTL description showed that it is justified and often advantageous to adopt a ADL-drived RTL processor synthesis methodology.

References

[1] Jörg Henkel. Closing the SoC Design Gap. *Computer*, 36(9), 2003.

[2] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA.* Kluwer Academic Publishers, 2002.

[3] A. Halambi and P. Grun and V. Ganesh and A. Khare and N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

[4] 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.

[5] G. Hadjiyiannis and S. Hanono and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. of the Design Automation Conference*, Jun. 1997.

[6] Tensilica. *http://www.tensilica.com*.

[7] Stretch. *http://www.stretchinc.com*.

[8] Target Compiler Technologies. *http://www.retarget.com*.

[9] Souvik Basu and Rajat Moona. High Level Synthesis from Sim-nML Processor Models. In *Proc. of the VLSI Design Conference*, 2003.

[10] A. Fauth and M. Freericks and A. Knoll. Generation of Hardware machine Models from Instruction Set Descriptions. In *Proc. of the IEEE Workshop on VLSI Signal Processing*, 1993.

[11] P. Mishra and A. Kejariwal and N. Dutt. Synthesis-driven Exploration of Pipelined Embedded Processors. In *Int. Conf. on VLSI Design*, Jan. 2004.

[12] ASIP Meister. *http://www.eda-meister.org*.

[13] M. Itoh and S. Higaki and J. Sato and A. Shiomi and Y. Takeuchi A. Kitajima and M. Imai. PEAS-III: An ASIP Design Environment. In *Proc. of the Int. Conf. on Computer Design (ICCD)*, Sep. 2000.

[14] CoWare/LISATek. *http://www.coware.com*.

[15] J.R. Brown. Chromatic scheduling and the chromatic number problem. *Management Science*, 19:456–463, 1972.

[16] F.T. Leighton. A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84:489–506, 1979.

[17] Luiz C. V. dos Santos and Jochen A. G. Jess. A Reordering Technique for Efficient Code Motion. In *Proceedings of the Design Automation Conference*, 1999.

[18] S. Gupta, N. Savoiu, S. Kim, N. Dutt, R. Gupta, and A. Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Proceedings of the Design Automation Conference*, 2001.

[19] Motorola. *68HC11: Microcontroller http://www.motorola.com*.

[20] T. Gloekler and S. Bitterlich and H. Meyr. ICORE: A Low-Power Application Specific Instruction Set Processor for DVB-T Acquisition and Tracking. In *Proc. of the ASIC/SOC Conference*, Sep. 2000.

[21] Schliebusch, O. and Chattopadhyay, A. and Steinert, M. and Braun, G. and Nohl, A. and Leupers, R. and Ascheid, G. and Meyr, H. *RTL Processor Synthesis for Architecture Exploration and Implementation.* Paris, France, Feb 2004.

[22] Synopsys. *Design Compiler http://www.synopsys.com/products/logic/design_compiler.html*.

[23] Synopsys. *DesignWare Components http://www.synopsys.com/products/designware/designware.html*.

[24] GNU. *GNU Development Chain for 68HC11 http://www.gnu-m68hc11.org/*.