

Applying Resource Sharing Algorithms to ADL-driven Automatic ASIP Implementation

E. M. Witte, A. Chattopadhyay, O. Schliebusch, D. Kammler
R. Leupers, G. Ascheid, H. Meyr

*Institute for Integrated Signal Processing Systems
RWTH Aachen University
52056 Aachen, Germany
witte@iss.rwth-aachen.de*

Abstract

Presently, Architecture Description Languages (ADLs) are widely used to raise the abstraction level of the design space exploration of Application Specific Instruction-set Processors (ASIPs), benefiting from automatically generated software tool suite and RTL implementation. The increase of abstraction level and automated implementation traditionally comes at the cost of low area, delay or power efficiency. The standard synthesis flow starting at RTL abstraction fails to compensate for this loss of performance. Thus, high level optimizations during RTL synthesis from ADLs are obligatory. Currently, ADL-based optimization schemes do not perform resource sharing. In this paper, we present an iterative algorithm for performing resource sharing on the basis of global data flow graph matching criteria. This ADL-based resource sharing optimization is performed over a RISC and a VLIW architecture and two industrial embedded processors. The results indicate a significant improvement in overall performance. A comparative study with manually written RTL code is presented, too.

1. Introduction

In recent times, the high system complexity coupled with an ever shrinking fabrication process have greatly increased the development costs for application-specific embedded processors, a key building block of System-on-Chip (SoC). Finding the best-in-class solution from the vast design space of embedded processors is a difficult task. The decreasing time-to-market aggravated the problem, therefore promoting a strong interest towards automated ASIP design approaches. There are two major classifications in the current automatic ASIP design approaches. While the first approach [29] tunes a template processor core to suit the particular application, the second one allows to model the processor using ADLs [16] [10] [13] [1]. In this paper, we focus on the the second approach. The term *RTL processor synthesis* refers to the automatic generation of an RTL processor description from an ADL description.

The approaches based on ADLs enable the designer to model the processor on a higher abstraction level than traditional RTL. The software tool suite associated with the processor, consisting of a C-compiler, a simulator, an assembler and a linker is automatically generated from the ADL description of the processor. The targeted application(s) can be profiled by the automatically generated software tools and a quick design convergence

is achieved. Finally, the RTL description is automatically generated from the ADL. Current state-of-the-art RTL processor synthesis, however, is yet to achieve the performance of the RTL model, which is manually written by an experienced designer. The high level architectural information required for optimization becomes extremely complex to exploit during gate-level synthesis. This situation demands the development of an ADL-based optimization framework. With this background, we identified, that the software-like description style of ADLs, while providing a convenient modelling style to the designer, introduces redundancy in terms of functional hardware resources. In high-level description languages, it is commonplace to concentrate on the algorithmic flow. In contrast, in the RTL description, it is usual to economically allocate the costly hardware with an explicit focus on parallelism. Thus, the ADL description style - while offering the designer an efficient and compact way to design the processor - poses a serious bottleneck against the generation of an optimized RTL description. The ideal optimization to treat this issue is to perform *ADL-based resource sharing*. With the above motivation, we targeted resource sharing optimization and built a general framework for high-level optimizations. We chose LISA [8] as the ADL for automatic generation of *optimized RTL*. The contribution of this paper is to present:

- An framework for ADL-based resource sharing.
- An ADL-based resource sharing algorithm.

We show the efficacy of our optimization framework on the basis of two different metrics. The first one is the comparison of area improvement and area-delay-product (AT-product) improvement of the automatically generated RTL description with ADL-based resource sharing w.r.t. automatically generated RTL description without ADL-based resource sharing. The second one is the comparison of area and AT-product of automatically generated RTL description with ADL-based resource sharing w.r.t. handwritten RTL description.

This paper is organized as follows: section 2 introduces the previous work in this domain. Section 3 describes the optimization framework and background for applying resource sharing. In section 4, generic sharing methods are explained. Section 5 discusses the resource sharing algorithm in detail. The results are analyzed in section 6. This paper ends with summary and outlook.

2. Related work

The assignment of a single resource to multiple non-concurrent operations is called *resource sharing*. To meet strict area requirements, resource sharing is aggressively performed during the synthesis of digital circuits. Here, we discuss the major approaches in resource sharing. Subsequently, current ADL-based optimization schemes are discussed.

2.1. Resource sharing approaches

Traditionally, the problem of resource sharing is coupled with *scheduling* and *resource allocation* [17]. In this work, we derive the scheduling from the inherent data-dependency of ADL-based description. Resource sharing can be performed for storage elements as well as for functional units. We restrict our focus to *resource sharing for functional units*, since for most of the ADLs it is usual to explicitly specify the number of storage elements.

Raje and Bergamaschi [21] classified resource sharing algorithms into *constructive* and *global* algorithms. In constructive algorithms [15], the sharing is considered for one element at a time. In global algorithms, the sharing decision is correlated with a global view of the data-path. Due to the advantage of considering the data-flow and structural similarity, the latter approach got more acceptance [6][7]. Usually, in global resource sharing algorithms, resources are mapped on compatibility graphs on which clique partitioning techniques are applied [30][4]. The clique partitioning problem is known to be NP-complete. Therefore, the quality of results obtained depends on the efficacy of the applied heuristic algorithm. In [21], a heuristic for global clique partitioning is proposed. This algorithm takes structural similarities, sharing order, false loop elimination and control logic effects in the limited periphery of the candidate resources into account. In another global approach, Brisk et al. proposed an algorithm that is based on the matching of data flow paths [7]. The algorithm in [11], proposed by Geurts et al, also performs the resource sharing on the basis of bi-partite graph matching. In order to focus on the timing problems that may occur when applying resource sharing, Bhattacharya et al. proposed an algorithm to optimize resource sharing results under timing constraints [5]. With this approach they are able to get almost the same area savings as previous approaches along with a significantly increased operational speed. More recently, Um et al [31] tried to integrate layout information during resource sharing.

2.2. ADL-based resource sharing

Here we limit our discussion among the prominent ADLs, which support automatic generation of RTL description. The RTL generator GO, from Target Compilers Technologies [28], is based on the language nML [10]. 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 sharing of the functional elements during RTL generation from nML. Sim-nML [19] [2], 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 resource sharing. The synthesis tool HGEN [9] generates Verilog code from an ISDL description. ADL-based resource sharing is not covered by the HGEN approach. Architectural concepts as used in MIMOLA [1] specify e.g. pipeline structures and behavioral aspects on a level similar

to Hardware Description Language (HDL) code. With such an approach, it is complicated to extract the instruction-set, thereby resulting in an inefficient software tool performance. The synthesis approach from MIMOLA includes a coarse-level module selection process, without any sophisticated approach to optimize resource usage. EXPRESSION [14] is an ADL, based on a combination of instruction-set and architecture description. The RTL generation from EXPRESSION does not address any optimization framework [18].

3. Resource sharing framework

In this section, we briefly introduce the ADL LISA [16], followed by the graph-based representation and an approximate cost model.

3.1. LISA background

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 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. The 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 *activations*, which refer to the execution of another LISA operation. For a LISA operation P , the set of children \mathcal{C}_P is 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 operation graph can be distributed over several pipeline stages.

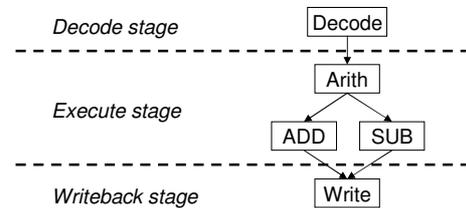


Figure 1: LISA Operation DAG

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” and “don’t care” (e.g. immediate bits of instruction encoding) bits or a nonterminal bit sequence referring to the coding of child LISA operations.

Activations: A LISA operation can *activate* other operations in the same or a later pipeline stage. The activation within the same pipeline stage does not cause any data-dependency between the activating and activated operations, making them implicitly concurrent.

Behavior Section: The behavior section of a LISA operation corresponds to the data path of the architecture. Inside the behavior description, 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. One behavior section can be conceptualized as one single *functional block*.

3.2. Exclusiveness and data flow

A unified representation of the processor is important in order to determine the *operational flow* and the *exclusiveness of functional blocks*, two key pieces of information for resource sharing.

3.2.1. Exclusiveness representation

We represent the exclusiveness between the functional blocks in a *conflict graph*. A conflict edge between two functional blocks indicates that those are not mutually exclusive.

$$A \odot B \cong \text{Block}_A \text{ conflicts with } \text{Block}_B$$

$$\mathcal{G}_{\text{conflict}} = \langle V, E_{\text{conflict}} \rangle$$

$$E_{\text{conflict}} = \{(x, y) | x, y \in V, x \odot y\}$$

The information about mutual exclusiveness among functional blocks is determined mainly from the instruction encoding of LISA operations, as shown in figure 2. This exclusiveness extraction on the ADL level is computationally much less complex and therefore, is more efficient compared to the exclusiveness detection on RTL level [23]. In figure 2, the operation *ALU* contains a coding field *Arith_Logical_Control* referring to three different child operations. These three child operations have different coding patterns and thus, are mutually exclusive. Intuitively, different coding patterns mean that the operations belong to different instructions. Since two different instructions guarantee temporal mutual exclusion (except instruction-level parallelism) therefore, the computational resources used by the two instructions are mutually exclusive. In figure 2, the operation *ALU* and the operation *Arithmetic* do share the same branch of the operation graph and they cannot be considered as mutually exclusive.

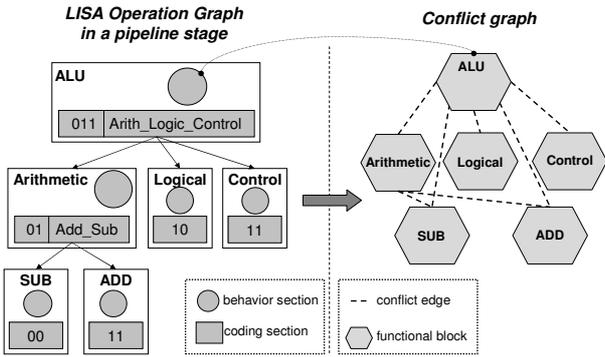


Figure 2: Exclusiveness extraction and representation

As shown in the figure, the *behavior* section of a LISA operation maps to a functional block. The exclusiveness information is propagated inside the functional block in order to allow an operator-level exclusiveness check during resource sharing. The exclusiveness detection inside a functional block relies on truth-table based static evaluation of conditions.

3.2.2. Data flow representation

On the high abstraction level, the data flow is captured by the dependency of LISA operations (also across the pipeline stages). Inside a single LISA operation, the *behavior* section guides the data flow. The behavior section of a LISA operation is converted into a pure, directed Data Flow Graph (DFG). The graph vertices of $G_{\text{DFG}} = \langle V_{\text{op}}, E_{\text{ic}} \rangle$ are the basic *operators* for data manipulation e.g. additions while edges represent the flow of unchanged data in form of *interconnections* of inputs and outputs.

Operators: The following list summarizes the basic classes of operators represented by graph vertices. This special choice of vertices allows us to represent the data flow information in a level between RTL and logic-level representation. In that way, our representation is close to Bergamaschi's *Behavioral Network Graph* [3].

- Commutative n -ary Operator, $n \geq 2$
- Noncommutative n -ary Operator, $n \geq 1$
- Read Access to Registers and Memories
- Write Access to Registers and Memories
- Read and Write Access to Array of Variable
- Multiplexer

Note that, unary operators are treated as a special case of Non-commutative n -ary operator.

Interconnections: Interconnections represent the data flow on symbol-level opposed to bit-level representations used in gate-level synthesis. The information about the data type transferred is given by an annotation to the interconnection. Bit range subscriptions are included into the interconnection information, too. A direct benefit of this approach is the possibility to encapsulate shift operators with a constant shift amount in bit ranges, thereby reducing the graph complexity.

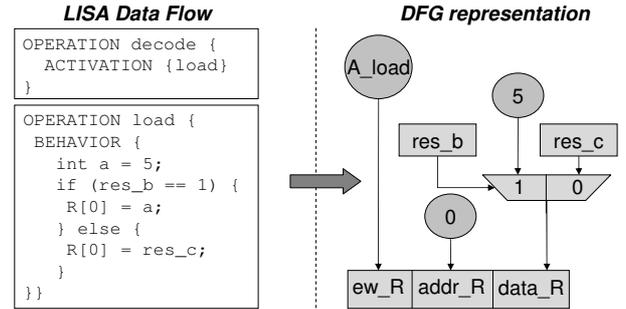


Figure 3: Example of data flow graph representation

The creation of the DFG from the plain C-code of a LISA operation's behavior section is shown in figure 3. As depicted there, the DFG is constructed after performing basic compiler-like optimizations. In this case, the constant value of the local variable a is propagated. For the read access to non-array registers e.g. res_c , we need not pass any address value. For the write access to a one-dimensional resource R , the write enable and the address value is set in the scope of the same vertex. The value for write enable is set to A_load , which indicates that this operation is being executed. Note, that this translation from C to a data flow produces a completely asynchronous DFG. Scheduling and pipelining is completely covered by the structure of the ADL model and not part of the DFGs.

Finally, each vertex of the DFG is annotated with the exclusiveness information obtained from the conflict graph. The resulting unified graph is called *Annotated Data Flow Graph* (ADFG).

3.3. Cost model

Resource sharing algorithms have to estimate and minimize the costs caused by sharing. These costs can be caused by additional multiplexers, which increase the timing delay and chip area and

thus lower the gain of sharing. Cost estimation is particularly important for incremental sharing algorithms that have to select the most promising pairs of sharing candidates iteratively. Using the DFG, a coarse approximation of the area and timing can be performed. Obviously, this approximation cannot take the implementation of arithmetic operations, the gate-level synthesis library or physical models into account.

Abstract Timing Approximation: In this work, a basic model for the delay estimation is used to avoid worse timing by improper sharing. The values for the delay are normalized to a virtual gate with a delay of one unit. All logical operations such as AND, OR, XOR and NOT are approximated by a delay of one unit. All other operations used in the data flow graph are modelled as multiples of basic gates. The implementation of the arithmetic operations, based on our experience with the gate-level synthesis tool, are considered and the delay is calculated on the basis of the input bit-width (W). Table 1 summarizes the models used for timing approximation. This table can be easily manipulated depending on the gate-level library to be used.

Table 1: Models for abstract timing approximation

Operators	Model	Timing Approximation
Bitwise (single-bit)	Single Gate	1
Addition, Subtraction	Carry Ripple Adder	$W + 2$
Comparison ($=$, \neq)	Tree Comparator	$\lceil \log(W) \rceil + 1$
Comparison ($>$, $<$)	Carry Ripple Adder	$W + 2$
Multiplication	Booth Multiplier, Carry Save Tree Adder	$5 + 3 * \lceil \log(W/2) \rceil + W$
Multiplexer	Tree of 2x1 multiplexers	$2 * W_{control}$

Abstract Area Approximation: When sharing two resources, multiplexer insertions might be necessary to share the different inputs. These multiplexers occupy the additional chip area A_{max} . If the area saving is smaller than A_{max} , the sharing will result in a negative gain. In many cases, a-priori assumptions can be made on the relations between the saved area by sharing two operators of a specific type and the area of necessary multiplexers. For example, no multiplexer implementation will be smaller than the area saved by sharing two bitwise operators. On the other hand, adder and multiplier implementations consume more area than the multiplexers inserted for the sharing. Based on these observations, in this work, a-priori assumptions are used for abstract area approximation.

4. Sharing methods

In this section, the generic sharing methods applicable for the ADL description of an ASIP are described. This is only a subset of analysis and optimization algorithms used for High Level Synthesis (HLS) or behavioral synthesis. The reason is the nature of cycle accurate ADL models which provide high level architecture information that renders several tasks like operation scheduling or clustering unnecessary. This section concentrates on the sharing of two different types of operators, namely non-commutative and commutative operators:

Non-commutative operators : Examples of non-commutative operators are subtraction, division, multiplexing etc. Two operators P_x and P_y can be shared if they implement the same function \mathcal{F} . When P_x and P_y are merged, there is no other possibility than sharing their inputs paired in the order given by their parameter position in function \mathcal{F} .

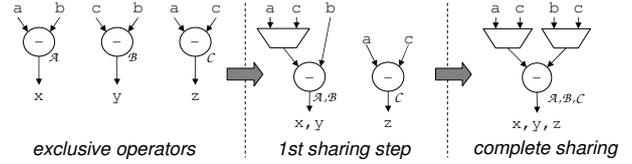


Figure 4: Non-commutative operator sharing

The example given in figure 4 shows three subtractions which have some common inputs. For efficient sharing, it is important to keep the number of multiplexers low i.e. to avoid introducing multiplexers for common inputs. In the first step of sharing, this happened for the input b . In the second sharing step, input c cannot be shared, since it is the first input of subtraction operator C .

Commutative operators : The sharing of two commutative operators P_x and P_y is possible if both operators implement the same function \mathcal{F} . The number of inputs does not need to be identical, because missing inputs can be completed by the identity element n with $\mathcal{F}(n, x_1, x_2, \dots) \equiv \mathcal{F}(x_1, x_2, \dots)$. Commutative operators also provide much more freedom in terms of sharing. The order of input operands for these operators can be permuted to derive the optimum grouping i.e. minimize multiplexer size.

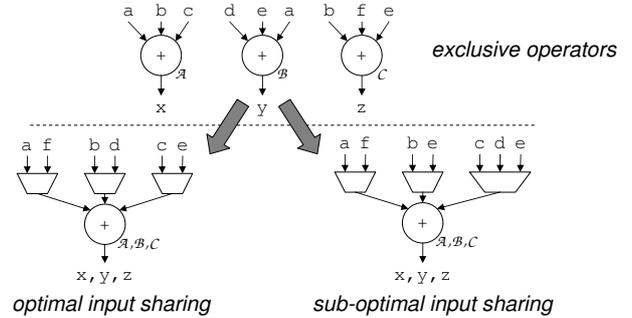


Figure 5: Commutative operator sharing

In the figure 5, an example of optimal and sub-optimal sharing of the same set of operators is shown. The optimum grouping of the input operands can be constructed by creating a conflict graph of input operands, where an edge represents concurrent usage of the different inputs in the same operator. In this work, a low-complexity greedy algorithm is applied to determine the near-optimal sharing.

5. Resource sharing algorithm

Aside from the necessity of efficient sharing methods of two given hardware resources, it is important to select the shareable resources with the most promising gain. In order to improve the overall performance, we adopted a global resource sharing approach with a cost-based graph matching method. In this section, first the *data flow graph matching criteria* are discussed. After that, the procedure for making sharing decision is elaborated.

5.1. Data flow graph matching criteria

The DFG matching criteria are the key to our resource sharing decisions. Currently, three following criteria are used.

Timing Difference : When sharing two operators, the critical path might get elongated significantly (figure 6). Therefore, the timing approximation given by the cost model is used to decide

whether the increase in the timing delay is acceptable or not. This maximum *acceptable* relative increase in delay is configurable.

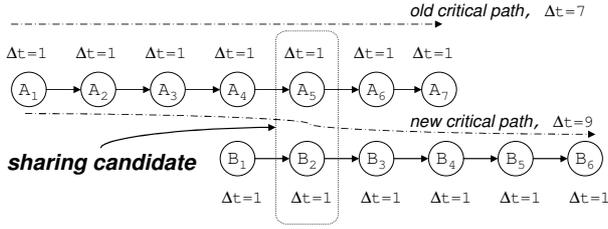


Figure 6: Possible worse timing as an effect of sharing

It is important to mention that this elongation of the critical path happens purely due to the limitation of the gate-level synthesis tool. Although the sharing is done between two mutually exclusive operators and therefore, those operators never execute in the same timing slot, yet a static analysis performed by the gate-level synthesis tool fails to recognize this.

Input Similarity: The number of common inputs between two candidate resources is defined as input similarity. The sharing of the pair of operators with the highest input similarity is desired. This criteria helps to reduce the number of multiplexers used during resource sharing.

Output Similarity: If the output of two operators serves as input for one and the same multiplexer, the sharing of both operators will remove the subsequent multiplexer and increase the gain of sharing. The number of common subsequent operators is defined as output similarity.

5.2. Sharing decision

The sharing algorithm first groups all operators within a data flow graph into sets $\mathcal{O}_{\mathcal{F}}$ of operators $O_{i,\mathcal{F}}$ representing the same function \mathcal{F} , e.g. a set of adders or a set of multipliers. With this strategy, the sharing problem is divided into smaller sets. For each set $\mathcal{O}_{\mathcal{F}}$ the pair with the best *rating* is selected for sharing, if there is any shareable pair. All pairs $\{(O_i, O_j) \mid O_i, O_j \in \mathcal{O}_{\mathcal{F}}\}$ are iteratively compared with the best shareable pair found so far. A pair $S = (O_i, O_j)$ is shareable under the following preferred conditions.

- The usage of both operators are mutually exclusive.
- Both operators have no data dependency introduced by a previous resource-sharing step, otherwise a *false loop* [24] would be created. *False loops* are not real loops, since in these loops the dependencies of control data for switches in the loop (e.g. multiplexers) make the circular data flow impossible. However, the information on the dependencies of the control signals cannot be extracted from the combinational logic during gate-level synthesis and the timing constraints are not met.
- The relative timing difference for O_i and O_j is less than the configured constraint.
- \mathcal{F} is a function such as *COMPARE*, *ADD*, *MUL*, *SUB*, *SHIFT*, which satisfies the a-priori assumption of abstract area approximation.

The rating is given by a relation defined for two pairs $S_1 = (O_i, O_j)$ and $S_2 = (O_k, O_l)$. The rating applies these prioritized comparisons in order to select the better pair.

1. If the input similarity for S_1 is higher than the input similarity for S_2 , S_1 is the better rated pair and vice versa. If

the input similarities are equal, proceed with further comparisons.

2. If the output similarity for S_1 is higher than the output similarity for S_2 , S_1 is the better rated pair and vice versa. If the output similarities are equal, proceed with further comparisons.
3. If the relative timing difference for S_1 is lower than the relative timing difference for S_2 , S_1 is the better rated pair and vice versa.

The above-mentioned heuristic rating is chosen for its simplicity and efficacy. The theoretically optimal solution, as discussed in section 2, is NP-complete. The complete sharing algorithm, of which the steps are already elaborated, is presented in the following pseudo-code.

```

01 // An Annotated Data Flow Graph (ADFG) contains
02 // exclusiveness and dataflow information.
03 //  $S_{type}$  is the set of operators of same type.
04 //  $S_{shareable}$  is the set of shareable operators.
05 //  $S_{best}$  is the pair of best shareable operators.
06 //
07 // The function call for each pipeline stage.
08 void ADL_Resource_Sharing(ADFGstage) {
09    $S_{type} = \text{Group\_Similar\_Operators}()$ ;
10   for each operator type  $S_{type_i}$  {
11      $S_{shareable_i} = \text{Find\_Shareable\_Operators}(S_{type_i})$ 
12     // loop termination
13     while ( $S_{shareable_i}$ ) is not empty {
14       Rate_Shareable_Operators( $S_{shareable_i}$ );
15        $S_{best_i} = \text{Get\_Best\_Shareable\_Operators}()$ ;
16       Share_Operators( $S_{best_i}$ );
17       Update_Data_Dependency(ADFGstage);
18       Update_Exclusiveness(ADFGstage);
19        $S_{shareable_i} = \text{Find\_Shareable\_Operators}(S_{type_i})$ 
20     }
21   }
22 }

```

The complexity of the above-mentioned algorithm relies on the number (n_{op}) of similar operators. The rating of shareable operators has to be done on every pair of operators (op_i, op_j). In worst case, where all the operators are shared, the inner loop (line 13) iterates over $(n_{op} - 1)$ times, leading to a complexity of $\mathcal{O}(n_{op}^3)$.

6. Results

The resource sharing optimization discussed in this paper is applied on 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. M68HC11 is a well-known micro-controller [20]. The ICORE architecture is dedicated for Terrestrial Digital Video Broadcast (DVB-T) decoding [12]. 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. The CoWare/LISATek RTL synthesis framework [22] allows automatic generation of RTL description from LISA. We applied our resource sharing optimization on top of this framework. All the RTL descriptions are synthesized with the Synopsys Design Compiler [25] and mapped to a technology library, which is based on a 0.18 μm process. The results for the quite simple architectures LTRISC and LTVLIW are presented in table 2. For both the architectures, we observed a modest area improvement due to the

simplicity of these models. The delay improvement is counter-intuitive and repeated synthesis runs showed that the delay measurements belong within the margin of the accuracy of the synthesis tool.

Table 2: Resource sharing: LTVLIW and LTRISC

Optimization phases	Area (K gates)	Delay (ns)	Area Reduction	AT-product Reduction
LTVLIW				
w/o resource sharing	7.78	4.12	-	-
with resource sharing	7.48	4.08	3.86%	4.79%
LTRISC				
w/o resource sharing	21.51	6.20	-	-
with resource sharing	19.21	6.12	10.7%	11.8%

To study the effects of optimization more in detail, we performed the resource sharing optimization for the ICORE architecture in two consecutive steps. First, we did the resource sharing inside each functional block and then we performed resource sharing across functional blocks. The initial optimization boundary was chosen to be a single LISA operation. The results in table 3 reveal some interesting facts. The resource sharing allowed a decrease of the critical path as the outputs of some shared operators were fed through a multiplexer which could be removed due to the output similarity criterion. However, this optimization goes along with a slight area increase due to buffer insertions during gate level synthesis. The multi-block resource sharing adds multiplexers at the operator inputs again and produces worse timing. A second reason for worse timing of multi-block resource sharing is to not consider the increment of delay due to the control signals of sharing multiplexer i.e. *exclusiveness signals*. This effect is particularly true for complex data-path, where the exclusiveness signals contribute significant delay. We will target this as our future enhancement of the cost model.

Table 3: Resource sharing: ICORE

Optimization phases	Area (K gates)	Delay (ns)	Area Reduction	AT-product Reduction
w/o resource sharing	37.98	6.11	-	-
with resource sharing (in a functional block)	41.93	5.12	-10.4%	7.49%
with resource sharing (across funct. blocks)	29.50	6.28	22.3%	20.2%

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 [26], running at clock frequencies up to 200 MHz (at 0.13 μm). An assembly-level M68HC11-compatible architecture called ISS68HC11 has been developed using LISA, with the goal to increase the instruction throughput. This is achieved by an optimized instruction coding, extending the 8 bit buses to 16 bit and using a pipelined architecture. As table 4 indicates, the architectural optimizations resulted in a cycle reduction of 62% with an overall speedup of up to 2.06.

Table 4: 68HC11 runtime (spanning tree application)

Architecture Version	Cycle-count	Speed (MHz)
M68HC11, 8-bit non-pipelined (hand-written RTL)	522608	200
ISS68HC11, 16-bit 3-stage pipelined (synthesized, with resource sharing)	195401	110..154

For obtaining a coarse comparison, we refer to the assembly

compatible hand written M68HC11 DesignWare component. Although the ISS68HC11 architecture is more complex, both architectures have the same requirements for low area and sufficient timing. The M68HC11 core CPU occupies between 15K and 30K gates depending on the speed, configuration and target technology [26]. On the other side, the RTL description of ISS68HC11 is automatically generated from LISA. The results are shown in table 5. It can be observed, that for this type of accumulator based architecture, the resource sharing gain is modest with about 13% area savings while the AT-product remains almost unchanged for sharing within single functional blocks and gets worse for sharing across blocks. As for the ICORE, the reason for the worsened timing is the inaccuracy of the current timing model concerning exclusiveness signals.

Table 5: Comparison with hand-written RTL

Architecture Version		Area (K gates)	Delay (ns)
68HC11	ISS68HC11, 16-bit 3-stage pipelined (synthesized, w/o resource sharing)	17.13	6.51
	ISS68HC11, 16-bit 3-stage pipelined (synth., single block sharing)	16.64	6.74
	ISS68HC11, 16-bit 3-stage pipelined (synth., sharing accross blocks)	14.87	9.06
	M68HC11, 8-bit non-pipelined (hand-written RTL)	15.00	5.00
ICORE	w/o resource sharing	37.98	6.11
	with resource sharing	29.50	6.28
	hand-written RTL	42.00	8.00

Table 5 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 resource sharing optimization enabled. The third version has been completely written manually in VHDL. The synthesis results show that the automatically generated RTL description can even be faster than the hand-written one. This is due to the difference in structural organization of the model. The hand-written ICORE VHDL description is organized with a centralized decoder. For the automatically generated description, the decoder is distributed over the entire pipeline, giving a better timing performance.

On the other hand, unoptimized automatically generated RTL description contains redundancy in terms of computational resources, resulting in area overhead. An ADL-based resource sharing optimization shares resources across the boundaries of functional blocks and results in much smaller area. Performing the same task by a designer manually is possible but would require a long design time and would be highly complex and error-prone.

Even though the cost model did not explicitly consider power consumption during resource sharing, the optimizations resulted in a power-efficient RTL description, too. For the ICORE architecture, we measured the power consumption using Synopsys Prime Power [27] (figure 7). The measurement reflects, that the power consumption is comparable for the Register File, since the number of registers is judiciously selected by the designer in LISA. Whereas, the functional units inside the pipeline consumed significantly more power without ADL-based resource sharing. This effect of resource sharing on power consumption is tightly related to the removal of redundant data path elements which even in idle state consume considerable power due to glitches e.g. during de-

coding. With our optimization, we are close to the power consumption for hand-written RTL description. Future work will target improvements of the sharing heuristic for steering directly the power reduction.

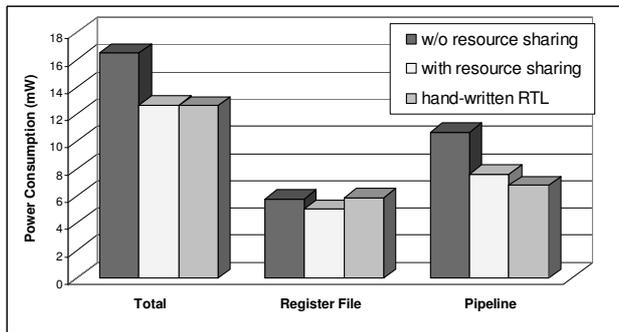


Figure 7: Effects of resource sharing on power : ICORE

7. Conclusion and future work

In this paper, we presented a framework and algorithm for resource sharing during ADL-driven RTL processor synthesis. The efficacy of this optimization is investigated over a RISC architecture, a VLIW architecture and two industrial embedded processors. The results indicate an area improvement of up to 22.3% compared to the basic architecture generated by RTL processor synthesis. A study on comparison with hand-written RTL description shows the effectiveness of ADL-driven resource sharing to match the performance of hand-written RTL description. The advantage of the approach mentioned in this paper, is to use the ADL-based processor design framework without any significant loss in the performance.

One of our future tasks is to make the optimization more controllable and scalable. Future work will include enhancing the cost model with more accurate estimation of delay, area and interconnect cost. Moreover, the applied heuristic of resource sharing will be augmented with further improved graph-matching criteria. We will also try various DFG-specific heuristics by applying weighted rating.

8. References

- [1] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA Language, Version 4.1. Reference Manual, Department of Computer Science 12, Embedded System Design and Didactics of Computer Science, 1994.
- [2] S. Basu and R. Moona. High Level Synthesis from Sim-nML Processor Models. In *VLSI Design*, 2003.
- [3] R. A. Bergamaschi. Behavioral network graph: Unifying the domains of high-level and logic synthesis. In *DAC*, pages 213–218, 1999.
- [4] R. A. Bergamaschi, R. Camposano, and M. Payer. Allocation algorithms based on path analysis. *Integration, the VLSI Journal*, pages 283 – 299, 1992.
- [5] S. Bhattacharya, S. Dey, and F. Breglez. Effects of resource sharing on circuit delay: an assignment algorithm for clock period optimization. *ACM Transactions on Design Automation of Electronic Systems.*, 3(2):285–307, 1998.
- [6] O. Bringmann and W. Rosenstiel. Resource sharing in hierarchical synthesis. *International conference on Computer-aided design*, pages 318–325, 1997.
- [7] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. *DAC 2004*, June 2004.
- [8] CoWare/LISATek. <http://www.coware.com>.
- [9] A. Fauth, 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.
- [10] A. Fauth, J. V. Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED&TC)*, Mar. 1995.
- [11] W. Geurts, F. Catthoor, and H. de Man. Quadratic zero-one programming based synthesis of application specific data paths. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 522–525. IEEE Computer Society Press, 1993.
- [12] T. Gloekler, 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.
- [13] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1997.
- [14] A. Halambi, P. Grun, V. Ganesh, A. Khare, 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.
- [15] C. Y. Hitchcock and D. E. Thomas. A method of automatic data path synthesis. *Proceedings of the 20th conference on Design automation*, pages 484–489, 1983.
- [16] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.
- [17] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1st edition, January 1994. ISBN: 0070163332.
- [18] P. Mishra, A. Kejariwal, and N. Dutt. Synthesis-driven Exploration of Pipelined Embedded Processors. In *Int. Conf. on VLSI Design*, Jan. 2004.
- [19] R. Moona. Processor Models for Retargetable Tools. In *IEEE International Workshop on Rapid System Prototyping*, 2000.
- [20] Motorola. *68HC11: Microcontroller* <http://www.motorola.com>.
- [21] S. Raje and R. A. Bergamaschi. Generalized resource sharing. *International Conference on Computer Aided Design*, 1997.
- [22] O. Schliebusch, A. Chattopadhyay, D. Kammler, R. Leupers, G. Ascheid, and H. Meyr. A Framework for Automated and Optimized ASIP Implementation Supporting Multiple Hardware Description Languages. In *Proc. of the ASPDAC - Shanghai, China*, Jan. 2005.
- [23] O. Schliebusch, A. Chattopadhyay, E. M. Witte, D. Kammler, G. Ascheid, R. Leupers, and H. Meyr. Optimization Techniques for ADL-driven RTL Processor Synthesis. In *IEEE International Workshop on Rapid System Prototyping*, 2005.
- [24] L. Stok. False loops through resource sharing. In *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 345–348. IEEE Computer Society Press, 1992.
- [25] Synopsys. *Design Compiler* http://www.synopsys.com/products/logic/design_compiler.html.
- [26] Synopsys. *DesignWare Components* <http://www.synopsys.com/products/designware/designware.html>.
- [27] Synopsys. *Prime Power* http://www.synopsys.com/products/power/primepower_ds.pdf.
- [28] Target Compiler Technologies. <http://www.retarget.com>.
- [29] Tensilica. <http://www.tensilica.com>.
- [30] C. J. Tseng and D. P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design*, pages 379–395, July 1986.
- [31] J. Um, J. Kim, and T. Kim. Layout-driven resource sharing in high-level synthesis. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 614–618. ACM Press, 2002.