Identification and Exploitation of Symmetries in DSP Algorithms

C.A.J. van Eijk[1] E.T.A.F. Jacobs[1] B. Mesman[1,2] A.H. Timmer[2]

[1] Design Automation Section, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

[2] Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands

Abstract

In many algorithms, particularly those in the DSP domain, certain forms of symmetry can be observed. To efficiently implement such algorithms, it is often possible to exploit these symmetries. However, current hardware and software compilers show deficiencies, because they cannot identify them. In this paper we, we propose two techniques to automatically detect and utilize symmetry. Both techniques introduce sequence edges between operations such that the feasibility of the scheduling problem is preserved, while the symmetry is broken. In combination with existing techniques for constraint analysis, this enhances the quality of compilers considerably, as is shown by benchmark results.

1. Introduction

For many applications, implementing algorithms efficiently is of utmost importance. This is particularly true in the field of digital signal processing (DSP), as DSP algorithms are usually part of an embedded system and therefore severely constrained with respect to timing, area and power. Despite the existence of high-level hardware and software compilers, the inner loops of such algorithms are often still optimized by hand, in order to obtain a solution that is efficient enough.

The main reason why compilers have a hard time to generate efficient code or hardware implementations is the combination of tight timing and resource constraints for DSP algorithms. Lately, research has therefore been focused on constraint analysis techniques, to be able to handle combinations of timing and resource constraints efficiently [1] [8][9][12][14]. Those analyses decrease the apparent scheduling freedom of a compiler, without removing any feasible schedules. So, the solution space is left unaltered, but the search space of a compiler is decreased. This is achieved by generating additional sequence edges between operations in a parallel representation of the application, and by narrowing the execution intervals (that is, the range of feasible start times) of the operations.

A major hurdle for the constraint analysis techniques described above are symmetric structures in algorithms. To illustrate what we mean by symmetry, some small examples are shown in Figure 1. In the example of Figure 1a, the two operations labeled with ‘rd’ are symmetric. Figure 1c shows an example that also exhibits symmetry, but in this case, it is clearly more difficult to express it. In the context of scheduling, an important consequence of symmetric structures is that they introduce a form of ‘redundancy’ in the search space: Schedules that only differ in the order in which symmetric operations are executed, can effectively be considered to be identical. Existing constraint analysis techniques do not utilize symmetry to decrease the apparent scheduling freedom, because they lack a theoretical framework to do so.

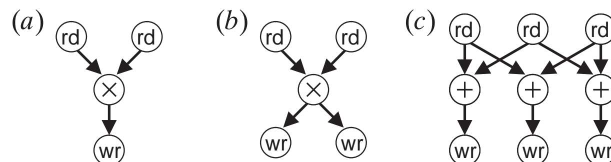


Figure 1. Examples of symmetry

In this paper we will show how one can use symmetry to make partial sequencing (or scheduling) decisions while preserving feasibility. We present a theoretical framework for capturing and utilizing symmetry, with the following advantages:

- It reduces the search space of a scheduler, without making a problem instance infeasible. Both heuristic schedulers and exact schedulers benefit from this. Heuristic schedulers will find a solution more easily, while exact schedulers will find a solution faster. For instance, a scheduler based on integer programming will need fewer variables.
- The symmetry analysis is complementary to existing constraint analysis techniques. It improves the search space reduction by removing symmetric, that is, identical, solutions. In contrast to the constraint analysis techniques, this analysis reduces the solution space, but guarantees at the same time that it does not make a problem instance infeasible.
- For approaches that test the feasibility of a problem instance or determine lower bounds for a schedule [2][10] [14][15], the analysis provides improved accuracy. It will be shown that the latency, throughput or resource requirements do not change due to the analysis.

Within the field of high level synthesis, the utilization of symmetry has received little attention. In practice, in many

cases the inner loops of DSP algorithms are still optimized by hand. In [6], it is shown how knowledge of symmetry in the datapath of a circuit can improve the efficiency of the integrated scheduling, allocation and binding approach of OSCAR. However, that work does not deal with symmetry present in the data flow graph. Our notion of 'symmetry' should not be confused with the notion of 'regularity' as for example used in [7]. Regularity refers to the repeated occurrence of patterns of operations, which is a more local property than symmetry.

In Section 2, we will start with some basic definitions. Section 3 discusses how symmetry can be modeled. The two subsequent sections describe our methods for exploiting symmetry. Section 6 gives empirical results and Section 7 concludes.

2. Definitions

An algorithm can be represented by a data flow graph (DFG). A DFG is a directed acyclic graph (V, E), where V is the set of operations, and $E \subset V \times V$ is the set of edges representing the data dependencies between the operations. Each operation has a certain type (e.g. addition or multiplication). The set of all operation types is denoted T_O. The function $\tau : V \rightarrow T_O$ assigns to each operation its operation type. The function $\delta : T_O \rightarrow \mathbb{N}^+$ associates a positive delay with each operation type.

A schedule φ is a function of type $V \rightarrow \mathbb{N}$ that assigns to each operation a start time. A schedule is called *feasible* if it does not violate the data dependencies, or any other constraint imposed on the schedule. In this paper we only formalize the data dependencies, because it is easy to see that many other important properties of a schedule are preserved by the transformations we use, such as the latency, the initiation interval, and the resource usage.

Definition 1
A schedule $\varphi : V \rightarrow \mathbb{N}$ meets the data dependencies iff for all edges $(u, v) \in E$:

$$\varphi(v) \geq \varphi(u) + \delta(\tau(u)) .$$

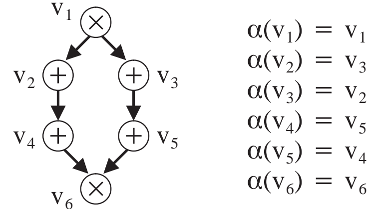
3. Modeling symmetry

In general, the concept of symmetry is strongly related to the property of an object that it does not change under a certain transformation. The kind of transformation that is relevant for capturing symmetry in a DFG is a relabeling of the operations such that the operation types and the data dependencies are preserved. Such a transformation is called an automorphism [11].

Definition 2
An *automorphism* is a bijective function $\alpha : V \rightarrow V$ such that for all $u, v \in V$:

- $\tau(v) = \tau(\alpha(v))$,
- $(u, v) \in E \Leftrightarrow (\alpha(u), \alpha(v)) \in E$.

To illustrate this definition, a simple example is shown in Figure 2. The symmetry that is clearly present in the shown DFG is captured by the automorphism α that exchanges v_2 and v_3, and also v_4 and v_5.



$\alpha(v_1) = v_1$
$\alpha(v_2) = v_3$
$\alpha(v_3) = v_2$
$\alpha(v_4) = v_5$
$\alpha(v_5) = v_4$
$\alpha(v_6) = v_6$

Figure 2. An example of an automorphism

We call two operations $u, v \in V$ *symmetric* if there exists an automorphism that maps u to v. The existence of an automorphism indicates the presence of symmetry. In fact, an automorphism can be used to transform a feasible schedule into another feasible schedule, as is shown in the following theorem and illustrated in Figure 3. This important result forms the basis for the technique presented in Section 4.

Theorem 1
Given a feasible schedule φ and an automorphism α. The schedule φ' defined by:

$$\varphi'(v) = \varphi(\alpha(v))$$

is also a feasible schedule.

The proof is given in Appendix A.

v	$\varphi(v)$
v_1	0
v_2	2
v_3	1
v_4	3
v_5	4
v_6	5

α: $v_1 \rightarrow v_1$, $v_2 \rightarrow v_3$, $v_3 \rightarrow v_2$, $v_4 \rightarrow v_5$, $v_5 \rightarrow v_4$, $v_6 \rightarrow v_6$

v	$\varphi'(v)$
v_1	0
v_2	1
v_3	2
v_4	4
v_5	3
v_6	5

Figure 3. Schedule transformation for Figure 2

A basic step in the methods described in the following two sections is the determination of an automorphism. The problem of deciding whether an automorphism exists for a DFG (and determining one if it exists) is closely related to the graph isomorphism problem for directed acyclic graphs. In general this is a difficult problem. For our method we use a well-known technique to iteratively partition the set of operations into equivalence classes such that operations that are not in the same equivalence class cannot be symmetric . This technique is e.g. also used for netlist comparison [3]. In our experience this technique is very efficient in practice for DFGs. Globally, it works as follows. First all operations are partitioned based on some local properties, such as the operation type and the number of predecessors and successors. Then this partition is iteratively refined by considering the neighborhood of the operations. Two operations only remain in the same class of the partition if the distribution of their direct predecessors over the equivalence classes is identical, and similarly for the direct successors. If this re-

sults in a partition in which each class contains at most two operations, an automorphism has been found. Otherwise, several possibilities have to be evaluated, resulting in a branch-and-bound approach in which the refinement process is also continued. A more detailed description of this technique is beyond the scope of the paper.

4. Breaking symmetry

In the previous section, it was shown how an automorphism can be used to transform a schedule while preserving feasibility (and also all properties involving latency, initiation interval, and resource usage). We will now explain how this transformation can be used to impose extra constraints on the DFG without excluding all feasible schedules. These constraints take the form of sequence edges between operations [8]. A sequence edge from an operation u to an operation v with integer weight w expresses the constraint that operation u should start at least w cycles before operation v. This means that any feasible schedule φ has to satisfy:

$$\varphi(v) \geq \varphi(u) + w.$$

The techniques proposed in this paper break the symmetry in a DFG by introducing sequence edges with weight zero between symmetric operations. Note that this still allows symmetric operations to be executed simultaneously, if sufficient resources are available. The extra constraints can significantly enhance the accuracy of existing constraint analysis techniques, as will be shown in Section 6.

Given an operation v and a set of automorphisms $A = \{\alpha_1, \ldots, \alpha_n\}$ that map v to another operation, i.e., for each $\alpha_i \in A$, $\alpha_i(v) \neq v$. With abuse of notation, we denote the set of operations to which v is mapped by A(v), i.e., $A(v) = \{\alpha_i(v) \mid \alpha_i \in A\}$. The following DFG transformation describes the introduction of sequence edges to break the symmetry.

Feasibility preserving transformation 1
Given an operation v, for each operation $u \in A(v)$, a sequence edge is introduced from u to v with weight zero.

For the DFG of Figure 2, we have four different possibilities to add a sequence edge. We can either introduce a sequence edge from v_2 to v_3, or from v_4 to v_5, or from v_3 to v_2, or from v_5 to v_4. The following theorem proves that Transformation 1 indeed preserves feasibility. It shows that if there exists a feasible schedule for the original DFG, then there also exists a feasible schedule that satisfies the additional sequence edges.

Theorem 2
If there exists a feasible schedule φ, then there also exists a feasible schedule φ' with the property that v is executed not later than any operation in the set A(v), i.e., for all $u \in A(v)$:

$$\varphi'(v) \leq \varphi'(u) .$$

The proof is given in Appendix A.

We propose a method that applies Transformation 1 to all operations in a DFG. For each operation v, we try to determine a set of symmetric operations by determining a set of automorphisms. To guarantee that only consistent choices are made when Transformation 1 is applied more than once, we use the following technique. Once we have introduced sequence edges for some operation v and continue with another operation, we only allow automorphisms that map v to v. For the example of Figure 2, this means e.g. that if first v_2 is considered, that, based on the automorphism shown in this figure, a sequence edge is introduced from v_2 to v_3. If subsequently operation v_4 is considered, then this automorphism is not used again, because it does not leave operation v_2 fixed. The resulting algorithm is shown in Figure 4.

```
void breakSymmetry (U ⊆ V)
{
  /* U is the set of operations that have not yet been
     analyzed; initially U equals V. */
  if (U = ∅)
     return;
  choose an operation v ∈ U
  forall u ∈ U\{v}
     if (an automorphism α exists such that α(v) = u and
         for all w ∈ V\U : α(w) = w)
        add a sequence edge from v to u with weight zero;
  breakSymmetry(U\{v});
}
```

Figure 4. Algorithm for breaking symmetry

5. Scheduling isomorphic sub-graphs

In many practical algorithms that exhibit symmetry, the symmetry is of a simple kind that can be characterized as follows: The automorphism identifies two non-overlapping isomorphic sub-graphs. The DFG in Figure 2 is an example of this, where the operations in the two sub-graphs are $\{v_2, v_4\}$ and $\{v_3, v_5\}$. For this type of symmetry we have an additional method, which handles the entire sub-graphs instead of individual operations. Therefore it can introduce more sequence edges than the method described in the previous section. E.g. for the example of Figure 2, it introduces a sequence edge from v_2 to v_3 as well as from v_4 to v_5 (the method of the previous section will introduce only one of these sequence edges). The method is based on the observation that when two isomorphic sub-graphs have to be scheduled, it is possible to order the execution of all their operations, such that each operation of one of the sub-graphs is executed not later than the corresponding operation in the other sub-graph. Before we explain this in more detail, we first give the conditions an automorphism has to satisfy for this method.

Definition 3

Given an automorphism $\alpha : V \rightarrow V$. Let V_{fix} denote the set of operations that are mapped to themselves, i.e., $V_{fix} = \{u \in V \mid \alpha(u) = u\}$. The automorphism α is said to *induce isomorphic sub-graphs* if all operations not in V_{fix} can be partitioned into two sub-sets V_1 and V_2 that satisfy the following two rules:

- for each operation $v \notin V_{fix}$, the operations v and $\alpha(v)$ are not in the same sub-set;
- for each edge $(u, v) \in E$ with $u, v \notin V_{fix}$, the operations u and v are in the same sub-set.

For a given automorphism α, it can be determined in a single traversal of the DFG whether it induces isomorphic sub-graphs and what the sets V_1 and V_2 are (by repeatedly assigning an arbitrary operation outside V_{fix} to V_1 and propagating this assignment using the two rules given in Definition 3). Note that Definition 3 is completely symmetric in the sets V_1 and V_2: the decision which set is which determines how the symmetry is broken.

Based on the sets V_1 and V_2, the following DFG transformation is defined.

Feasibility preserving transformation 2

For each operation $v \in V_1$, a sequence edge with weight zero is introduced from v to $\alpha(v)$.

The following theorem proves that the above transformation indeed preserves feasibility. It shows that if there exists a feasible schedule for the original DFG, then there also exists a feasible schedule that satisfies the extra sequence edges introduced by Transformation 2.

Theorem 3

Given an automorphism $\alpha : V \rightarrow V$ that induces two isomorphic sub-graphs with the sets of operations V_1 and V_2. If a schedule $\varphi : V \rightarrow \mathbb{N}$ is feasible, then so is the schedule $\varphi' : V \rightarrow \mathbb{N}$ defined by:

$$\varphi'(v) = \begin{cases} \varphi(\alpha(v)) & \text{if } v \in V_1 \wedge \varphi(v) > \varphi(\alpha(v)) \\ \varphi(\alpha(v)) & \text{if } v \in V_2 \wedge \varphi(v) < \varphi(\alpha(v)) \\ \varphi(v) & \text{otherwise} \end{cases}.$$

The proof is given in Appendix A. The construction of φ' for the example of Figure 2 is shown in Figure 5.

v	$\varphi(v)$		v	$\varphi'(v)$
v_1	0	→	v_1	0
v_2	2	↘	v_2	1
v_3	1	↗	v_3	2
v_4	3	→	v_4	3
v_5	4	→	v_5	4
v_6	5	→	v_6	5

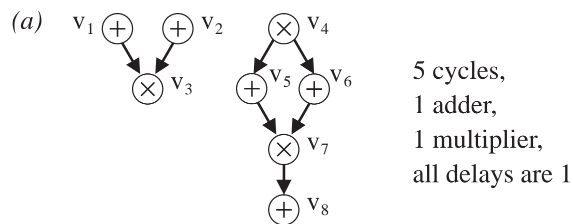
Figure 5. Schedule φ' for the example of Figure 2

Using a similar approach as presented in the previous section, Transformation 2 is applied as follows. For each operation v in the DFG, we try to extract automorphisms that map v to other operations. For each automorphism that satisfies Definition 3, Transformation 2 is applied. To guarantee that no inconsistent choices are made when applying the transformation multiple times, the sub-graphs of each transformation are stored; a new sub-graph is excluded if it has an overlap with an existing sub-graph without completely containing it. The resulting algorithm is similar to the algorithm shown in Figure 4, except that the condition in the if–statement also reflects the requirements stated in Definition 3, and that if this condition holds, for each operation $v_1 \in V_1$, a sequence edge is added to $\alpha(v_1)$.

6. Experimental results

We have implemented the methods proposed in this paper in the high level synthesis system FACTS under development at the Eindhoven University of Technology. In this section we report on the experiments we have performed with algorithms incorporating various degrees of symmetry. All experiments are run on a HP9000/879 workstation with a 180 Mhz PA-8000 processor.

The first experiment illustrates how the detection of symmetry can improve the accuracy of existing constraint analysis techniques. In particular, we consider the operation execution interval (OEI) analysis of [12][14]. The example DFG is shown in Figure 6a. Both the methods described in this paper have the same effect for this example, namely the creation of a sequence edge from v_1 to v_2, and also one from v_5 to v_6. To visualize the impact of these sequence edges, we focus on the execution interval of each operation. Figure 6b shows the intervals that are determined when our symmetry analysis is not used. The grey and diagonally striped regions together depict the ASAP-ALAP interval, and the grey region represents the execution interval after the OEI analysis.



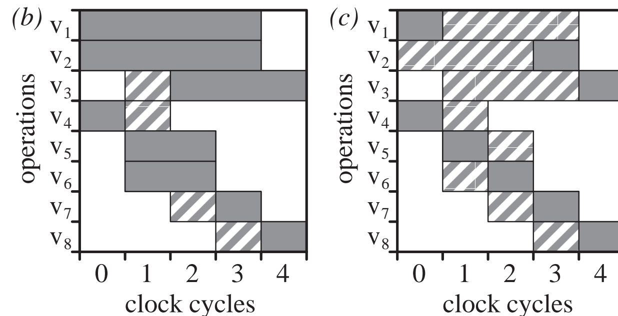


Figure 6. An example: (a) the data flow graph, (b) the results of constraint analysis, and (c) the results of symmetry detection and constraint analysis

Figure 6c shows the results of this same analysis after the introduction of the sequence edges. Note that as a result, the execution interval of operation v_3, which is not symmetric with any other operation, is reduced as well.

In the second experiment, we evaluate the proposed methods for utilizing symmetry on the well-known fast discrete cosine transform (FDCT) benchmark. For this example, the method for breaking symmetry as described in Section 4 is able to introduce 10 sequence edges, whereas the method for scheduling isomorphic sub-graphs creates 26 sequence edges.

The results are listed in Table 1 for various instances of FDCT, each with a latency constraint denoted by the number in the name of the instance, and a certain set of execution units (the exact numbers can be found in [13][14]). The postfix 'nf' denotes the instances where the combination of the latency constraint and the set of execution units results in infeasibility. The table shows the average freedom of the operations, which equals the average length of the execution interval minus one (as defined in [14]). The columns list the average freedom as determined by respectively ASAP-ALAP analysis, OEI analysis, and the method for scheduling isomorphic sub-graphs followed by OEI analysis. The last column shows the average improvement of the OEI analysis that results from the symmetry detection algorithm. In all instances except 'FDCT 9', the detection of symmetry results in a more accurate determination of the average freedom. For 'FDCT 9nf', the detection of symmetry allows the OEI analysis to determine that this instance is infeasible. The run-time for each instance (including all analyses) is less than 0.2 (s).

Table 1. Constraint analysis results for FDCT

instance	average freedom of operations			improv.
	ASAP-ALAP	OEI anal.	symm. + OEI an.	
FDCT 8	1.43	0.57	0.43	25%
FDCT 9nf	2.43	1.57	infeas.	100%
FDCT 9	2.43	1.76	1.76	0%
FDCT 10nf	3.43	2.90	2.59	11%
FDCT 10	3.43	3.23	2.76	15%
FDCT 11	4.43	2.81	1.50	47%
FDCT 13	6.43	6.14	5.64	8%
FDCT 14	7.43	6.67	5.64	15%
FDCT 18	11.43	9.52	8.62	9%
FDCT 26	19.43	17.00	15.33	10%
FDCT 34	27.43	23.00	17.90	22%

Figure 7 shows a part of the FDCT example where isomorphic sub-graphs are found. It is interesting to observe that our method is capable of detecting isomorphic sub-graphs within isomorphic sub-graphs.

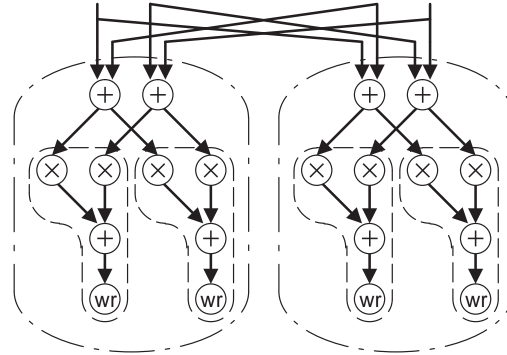


Figure 7. Some of the isomorphic sub-graphs found in FDCT (when +'s and –'s have the same operation type)

For the FDCT example, the method of Section 5 is always more effective than the method of Section 4, with average improvements of respectively 24% and 17% over the OEI analysis without symmetry detection. The average improvement of OEI analysis and the method of Section 5 over ASAP-ALAP analysis is 39%, which is very high considering the fact that in all feasible examples, the actual average freedom is larger than zero.

In the third experiment, we consider the loop bodies from an industrial example that combines a FFT with differential modulation [4]. We also consider the inverse discrete cosine transform with 11 multiplications from [5]. For each example, we determine two numbers: the first one is the average freedom of the operations. The second one is the depth of the searchtree of a branch & bound scheduler [13] [14] at the place where it finds a feasible schedule. The lower this number, the easier it is to find a schedule. The results are shown in Table 2. The first two columns list the name of the example and then the number of operations, the latency and the initiation interval. Then the average freedom and the depth of the searchtree are given without and with the use of the method from Section 5.

Table 2. Results for industrial examples

example	#ops/lat/II	no symm. det.		symm. det.	
		avg.	depth	avg.	depth
loop 1	15/6/2	1.00	5	0.60	4
loop 2	12/6/2	0.75	4	0.25	2
loop 3	30/12/4	6.27	28	1.97	17
loop 4	43/14/5	4.60	34	3.21	23
loop 5	43/14/5	4.60	30	2.14	16
loef11	40/14/11	6.35	44	6.04	31

The results clearly show that this method improves the accuracy of the scheduler significantly. The impact of the method can also be clearly observed by considering examples where the combination of the time and resource constraints together cause infeasibility. Table 3 shows the run-time needed by a branch & bound scheduler to

determine infeasibility. The column 'b&b' shows whether the scheduler is invoked to prove infeasibility.

Table 3. Results for infeasible examples

example	ops/lat/II	no symm. det.		symm. det.	
		b&b	time	b&b	time
FDCT 9nf	42/9/9	yes	0.1 (s)	no	0.1 (s)
FDCT 10nf	42/10/10	yes	>2hrs	yes	369 (s)
loop 4	43/14/4	yes	6.6 (s)	yes	0.4 (s)
loop 5	43/14/4	yes	5.0 (s)	no	0.6 (s)

To evaluate the run-time performance of symmetry detection on larger DFGs, we have applied it to scalable data flow graphs that exhibit a lot of symmetry. Our experiments confirm the efficiency of the approach. For example, a DFG with about 750 operations is analyzed in 12 (s).

7. Conclusions

In this paper, we presented a theoretical framework for identifying symmetries in DSP algorithms. We showed that these symmetries can be exploited by introducing extra sequence edges in the DFG while preserving the feasibility of the scheduling problem. These sequence edges effectively reduce the number of schedules that are equivalent modulo symmetry. The presented theory is proved in a general setting, so that the work is applicable in the context of hardware compilation as well as code generation.

We proposed two methods for utilizing symmetry. The first one is based on the idea of detecting symmetries for the individual operations, while the second one considers isomorphic sub-graphs, typically resulting in more sequence edges. However, the first method has the advantage that it can handle a wider range of symmetries because it imposes weaker conditions on the automorphism capturing the symmetry. Both methods are complementary to existing constraint analysis techniques, because unlike these techniques, they focus on preserving the feasibility of the scheduling problem rather than preserving all feasible schedules. The experimental results show that the identification of symmetry can result in a more accurate determination of the scheduling freedom.

Further research will focus on the integration of the two methods proposed in this paper. Furthermore we want to investigate how the presented methods can be applied *during* scheduling. Once a certain operation is scheduled, the symmetry analysis can essentially ignore it, possibly resulting in more symmetries for the operations that still have to be scheduled. We expect that this will further improve the applicability of our symmetry analysis.

Acknowledgements

The authors would like to thank Hans Cuypers and Bart Theelen for some helpful discussions on symmetry and computational group theory.

References

[1] S. Chaudhuri, R.A. Walker, and J.E. Mitchell, "Analyzing and Exploiting the Structure of the Constraints in the ILP Approach to the Scheduling Problem", IEEE Trans. on VLSI Systems 2(4), pp. 456–471, December 1994.

[2] S. Chaudhuri, S.A. Blythe, and R.A. Walker, "A Solution Methodology for Exact Design Space Exploration in a Three Dimensional Design Space", IEEE Trans. on VLSI Systems 5(1), pp. 69–81, March 1997.

[3] C. Ebeling, and O. Zajicek, "Validating VLSI Circuit Layout by Wirelist Comparison", Int. Conf. on Computer-Aided Design, pp. 172–173, 1983.

[4] J.A. Huisken, et al., "A Power-Efficient Single-Chip OFDM Demodulator and Channel Decoder for Multimedia Broadcasting", Proc. Solid-State Circuits Conference, pp. 40–41, 1998.

[5] C. Loeffler, A. Ligtenberg, and G.S. Moschytz, "Practical Fast 1D-DCT Algorithms with 11 Multiplications", Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing, pp. 988–991, 1989.

[6] P. Marwedel, et al., "A Technique for Avoiding Isomorphic Netlists in Architectural Synthesis", Proc. European Design & Test Conf., p. 600, 1996.

[7] R. Mehra, and J. Rabaey, "Exploiting Regularity for Low-Power Design", Proc. Int. Conf. on Computer-Aided Design, pp. 166–172, 1996.

[8] B. Mesman, et al., "Constraint Analysis for Code Generation", Proc. Int. Symp. on System Synthesis, 1997.

[9] B. Mesman, et al., "A Constraint Driven Approach to Loop Pipeling and Register Binding", Proc. Conf. on Design, Automation and Test in Europe, 1998.

[10] A. Sharma, and R. Jain, "Estimating Architectural Resources and Performance for High-Level Synthesis", Proc. 30th Design Automation Conference, pp. 355–360, 1993.

[11] C.C. Sims, "Computation with Finitely Presented Groups", Cambridge University Press, 1994.

[12] A.H. Timmer, and J.A.G. Jess, "Execution Interval Analysis under Resource Constraints", Proc. Int. Conf. on Computer-Aided Design, 1993.

[13] A.H. Timmer, and J.A.G. Jess, "Exact Scheduling Strategies based on Bipartite Graph Matching", Proc. European Design & Test Conf., pp. 42–47, 1995.

[14] A.H. Timmer, "From Design Space Exploration to Code Generation", Ph.D. thesis, Eindhoven University of Technology, 1996.

[15] M. Xu, and F.J. Kurdahi, "Layout-Driven High Level Synthesis for FPGA Based Architectures", Proc. Conf. on Design, Automation and Test in Europe, 1998.

Appendix A

Proof of Theorem 1

Consider a data dependency $(u, v) \in E$. Because α is an automorphism and $(u, v) \in E$, also $(\alpha(u), \alpha(v)) \in E$. Because φ meets all data dependencies, we have:

$$\varphi(\alpha(v)) \geq \varphi(\alpha(u)) + \delta(\tau(\alpha(u))) . \quad (1)$$

It follows from Definition 2 that $\tau(\alpha(u))$ equals $\tau(u)$, and therefore Equation 1 can also be written as:

$$\varphi(\alpha(v)) \geq \varphi(\alpha(u)) + \delta(\tau(u)) . \quad (2)$$

Because $\varphi'(v) = \varphi(\alpha(v))$, it follows that:

$$\varphi'(v) \geq \varphi'(u) + \delta(\tau(u)) , \quad (3)$$

which shows that also schedule φ' satisfies the data dependency (u,v). Because this reasoning holds for all the data dependencies, it follows that φ' meets all the data dependencies. Furthermore, because we have only exchanged the start times of operations having the same operation type, the schedule φ' results in the same latency, initiation interval, and resource usage as φ. Therefore we conclude that φ' is a feasible schedule. ■

Proof of Theorem 2

Assume that there is an operation $u \in A(v)$ such that $\varphi(u) < \varphi(v)$. Because $u \in A(v)$, there is an automorphism $\alpha_i \in A$ that maps v to u. If we consider the schedule φ_i' defined by:

$$\varphi_i'(v) = \varphi(\alpha_i(v)) , \quad (4)$$

then v is moved to an earlier time step. We can repeat such transformations until the condition of the theorem holds, because the number of possible start times is finite and after each transformation, the start time of v is decreased. ■

Proof of Theorem 3

Similar to the proof of Theorem 1, we first consider the data dependencies. We will show that if φ fulfils a data dependency, then so does φ'. Consider the data dependency $(u, v) \in E$. We do a case analysis on the membership of u and v of the sets V_{fix}, V_1 and V_2, as shown in the following diagram:

u \ v	V_{fix}	V_1	V_2
V_{fix}	(a)	(b)	(b)
V_1	(b)	(c)	X
V_2	(b)	X	(d)

Note that two cases cannot appear because of the third rule in Definition 3.

Case (a): $u, v \in V_{fix}$. This case is straightforward, because the start times of u and v are not changed.

Case (b): Assume that $u \in V_{fix}$ and $v \in V_1$. Because α is an automorphism that satisfies Def. 3, we know that $(u, \alpha(v)) \in E$ and $\alpha(v) \in V_2$. For schedule φ, we have:

$$\varphi(v) \geq \varphi(u) + \delta(\tau(u)) ,$$
$$\varphi(\alpha(v)) \geq \varphi(u) + \delta(\tau(u)) .$$

Therefore, these data dependencies cannot be violated by exchanging the start times of v and $\alpha(v)$. The other three sub-cases follow the same line of reasoning.

Case (c): Assume that $u, v \in V_1$. If $\varphi(u) \leq \varphi(\alpha(u))$ and $\varphi(v) \leq \varphi(\alpha(v))$, the data dependency (u,v) is not violated because the start times of u and v are not changed. If $\varphi(u) > \varphi(\alpha(u))$ and $\varphi(v) > \varphi(\alpha(v))$, the data dependency (u,v) is not violated because the start times of u and v with schedule φ' equal those of $\alpha(u)$ and $\alpha(v)$ with schedule φ, and schedule φ satisfies the data dependency $(\alpha(u), \alpha(v))$. If $\varphi(u) > \varphi(\alpha(u))$ and $\varphi(v) \leq \varphi(\alpha(v))$, then the start time of u decreases while the start time of v remains unchanged. This obviously cannot cause a violation of the data dependency (u,v). This leaves the case that $\varphi(u) \leq \varphi(\alpha(u))$ and $\varphi(v) > \varphi(\alpha(v))$. We have to show that:

$$\varphi(\alpha(v) \geq \varphi(u) + \tau(\delta(u))) .$$

This follows directly from the fact that:

$$\varphi(\alpha(v)) \geq \varphi(\alpha(u)) + \tau(\delta(\alpha(u))) ,$$

and $\delta(\tau(u)) = \delta(\tau(\alpha(u)))$ and $\varphi(u) \leq \varphi(\alpha(u))$.

Case (d): The proof is analogous to the proof of case (c).

Because we only exchange the start times of symmetric operations which have the same operation type, the schedule φ' results in the same latency, initiation interval, and resource usage as φ. Therefore we conclude that φ' is a feasible schedule. ■