

Module Selection and Scheduling using Unrestricted Libraries

Adwin H. Timmer^{*}, Marc J. M. Heijligers^{*}, Leon Stok^{**}, Jochen A. G. Jess^{*}

^{*}Eindhoven University of Technology, Department of Electrical Engineering,
Design Automation Section, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

^{**}IBM T.J. Watson Research Center,
P.O. Box 218, Yorktown Heights, NY 10598, USA

Abstract

Most high-level synthesis schedulers are only capable of mapping an operation to one specific module type. To ensure a full design space exploration, a synthesis system should however select freely from a library containing modules with a large variety in delay, area and so on. A novel module selection and scheduling approach is presented, which allows the full use of such unrestricted libraries. Extensive benchmark results show very fast running times and optimal solutions. Hence this approach clearly illustrates the advantages of synthesis tools which can fully cope with unrestricted libraries, as they lead to designs with less module area.

1. Introduction

Many high-level synthesis systems are being developed [McFa90], which synthesize a data path from a description represented by a data flow graph (DFG). In a time constrained design, operations are mapped on hardware modules selected by the system. To ensure a full design space exploration, a system should select freely from a library containing modules with a large variety in delay, area and so on. A novel module selection and scheduling approach is presented, which allows the full use of such *unrestricted* libraries. We will focus on the module selection, which is modelled as a very tight mixed integer linear programming (MILP) problem with only a few integer variables.

First we will present some definitions and a module library classification. The classification is used in section 3 to show the limited capabilities of earlier module selection and scheduling algorithms. In section 4 our new approach is outlined and results are presented in section 5.

2. Definitions and library classification

A DFG is a tuple (V, E) , where V is the set of nodes (operations) and E the set of edges representing dependencies between operations. T is the set of available cycle steps. $Type$ is the set of operation types and $\tau: V \rightarrow Type$ is the mapping from an operation to its type. L is a set of module

types and $\mu: P(Type) \rightarrow P(L)$ is a mapping from operation types to module types ($P()$ is the power set of). $\mu(ts)$ returns module types which can implement operations with a type from $ts \in P(Type)$. $c: L \rightarrow \mathbb{R}$ describes the cost of a module. $n: L \rightarrow \mathbb{N}$ returns the number of selected modules of a type.

$d: L \rightarrow \mathbb{N}^+$ describes the number of cycles a module type needs to execute an operation. $dii: L \rightarrow \mathbb{N}^+$ returns the data introduction interval (DII) of a module type. The DII is the minimal number of cycles required between the data arrivals for two executions. For simplicity we assume that the delays of a module are equal for all its operation types (the same holds for the DIIs); our method is however not restricted to the assumption made.

In a *class 1* library, each operation can be implemented by only one module type. In a *class 2* library, each module can implement only one operation type. For a *class 3* (or *unrestricted*) library no restrictions for library classes hold.

3. Previous work

Systems with class 1 schedulers can be adapted for class 3 libraries; [Jain92] and [Chen91] describe approaches to derive a class 1 library from a class 3 library by evaluating an estimation of the total area for each possible class 1 library. Our approach can be used in a similar way, and is far more accurate (see section 4.1.1). In [Ishi91] an approach has been given which tries to solve the scheduling, allocation and a class 2 module selection simultaneously with a hypergraph model. A class 3 library can not be incorporated because of restrictions of the hypergraph model. All these approaches do not explore the design space completely.

IP schedulers can be extended to cover class 3 libraries. The node packing modelling [Gebo92] seems to provide an efficient method to obtain optimal results. However, *time constrained* IP schedulers including *class 3* libraries are not time efficient. For *time constrained* scheduling, the model is no longer strictly a node packing problem. Also the complexity of the number of IP variables and constraints increases if a *class 3* library is used instead of a class 1 library.

In [Gutb91] and [Rama91] two list based class 3 schedulers are given, which are respectively resource and time constrained. Both try to estimate the influence of each pos-

sible mapping of operations, which can become very time consuming. The heuristic of [Rama91] is also rather imprecise, as it gives equal probability to each mapping of an operation to an applicable module type in the library.

4. Module selection & scheduling in the NEAT system

In the NEAT (New Eindhoven Architectural synthesis Toolbox) system, a time constrained scheduler is derived from a resource constrained list scheduler (see figure 1). First an initial module selection is made, which is a lower bound estimation of the module set with minimal area. The list scheduler tries to meet the time constraint with this selection. If the scheduler does not succeed, the module selection has to be reviewed until a correct selection has been made. So the problem is decomposed in two steps to deal efficiently with the design alternatives. In the first (module selection) step a global decision is made, while in the second (scheduling) step local decisions are made.

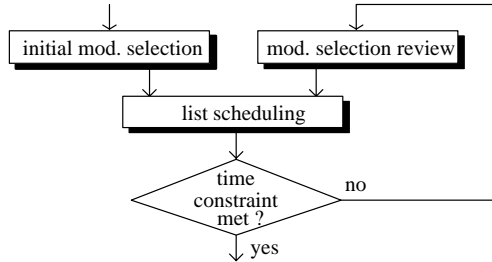


Fig 1: Time constrained NEAT-scheduler.

4.1. Module selection

The objective of the module selection is to select a module set with minimal area and is realized using MILP. The main problem with IP schedulers is that all precedence relations of a DFG appear explicitly as constraints, and that all variables have to be integral. Not all precedence relations are taken into account for the module selection, because the exact schedule is determined later on. Except for the variables denoting the number of selected modules, no variables have to be integral in our formulation. So the number of (integer) variables and constraints is kept limited. The time efficiency of solving this MILP problem depends upon the size of the library and not on the size of the DFG! In the following subsections some design characteristics are discussed, which lead to very tight and accurate module selection constraints.

4.1.1. Distribution of operations

Consider the example in figure 2 with a cycle budget of 4. [Jain92] and [Chen91] estimate the number of modules needed from a class 1 library, while assuming that all cycles are available to perform any operation (so no DFG dependencies are taken into account). There are 4 additions, and

an adder can perform 4 additions within the time constraint, so the result of their estimation is 1 adder. Now consider the execution intervals, which partly incorporate dependencies:

DEFINITION 1 [EXECUTION INTERVAL].

EI: $V \rightarrow [T, T]$ is a function which returns the interval in which an operation can be scheduled. This interval is calculated by determining the ASAP and ALAP values of the DFG nodes, while each operation in the DFG is mapped on the module with the smallest possible delay.

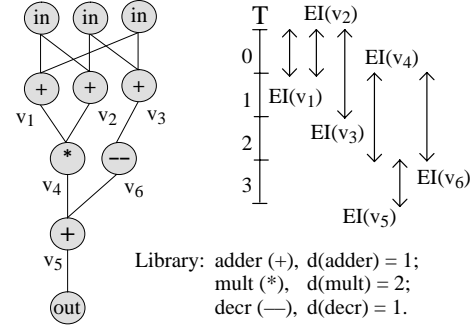


Fig 2: Example DFG, library and intervals.

The execution intervals of v_1 , v_2 and v_3 show that 3 additions must be executed within 2 cycles, so at least 2 adders are needed. These 3 intervals form a so called 'distribution interval'. Such a distribution interval denotes overlapping execution intervals. The example shows that the notion of distribution intervals leads to a more accurate estimation than the approaches of [Jain92] and [Chen91].

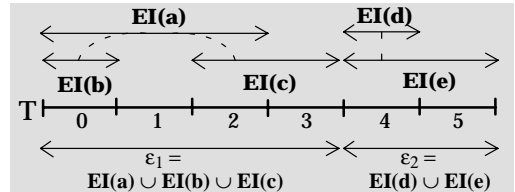


Fig 3: Execution and distribution intervals.

In figure 3, EI(a), EI(b) and EI(c) are in the same distribution interval (while EI(e) is only in the same distribution interval as EI(d)), because there is a 'path' from EI(b) to EI(c) through EI(a); this is denoted by the dashes. The formal description of distribution intervals is given below.

DEFINITION 2 [DISTRIBUTION INTERVALS].

Let $ts \in P(\text{Type})$ and $v_1, v_2 \in V$ with $\tau(v_1), \tau(v_2) \in ts$. The Boolean function $\text{group}(v_1, v_2)$ is true if: $\text{EI}(v_1) \cap \text{EI}(v_2) \neq \emptyset$ (so group is reflexive and symmetric), or if there is a $v_3 \in V$, $\tau(v_3) \in ts$ for which $\text{group}(v_1, v_3)$ and $\text{group}(v_2, v_3)$ are true (i.e. group is transitive). Group is an equivalence relation between operations of V with an operation type from ts . Let $K(ts)$ be the set of partitions given by group on all $v \in V$, $\tau(v) \in ts$. $\epsilon: P(\text{Type}) \rightarrow P[T, T]$ is the function which returns a set of disjoint distribution intervals: $\epsilon(ts) = \{ \bigcup_{v \in k} \text{EI}(v) \mid k \in K(ts) \}$.

The constraints try to enforce the selection of enough module 'capacity' to perform the operations in all the distribution intervals. It is clear that for each (set of) operation type(s) only the interval with the highest average of operations per cycle step is of interest. This interval is called the *had-interval* (highest average distribution interval). All other intervals normally need equal or less module capacity, and are therefore not considered in the MILP formulation.

To formulate the constraints, the 'number of preliminary mappings' and the 'capacity' of a module type have to be defined. The capacity states the number of operations a module can execute within an interval. As several module types can be selected for the operations in an interval, the number of operations mapped on each type must be given. These numbers are preliminary, as the exact mappings are determined by the scheduler. The exact definitions are:

DEFINITION 3 [NUMBER OF PRELIMINARY MAPPINGS].

$m: L \times P(\text{Type}) \times \text{Type} \rightarrow \mathbb{R}$ is a function describing the number of preliminary mappings to a module type in a had-interval; $m(\text{alu}, \{+, *, -, +\})$ is the number of additions mapped to the module type alu in the had-interval of the operation type set $\{+, *, -\}$.

DEFINITION 4 [CAPACITY OF A MODULE TYPE].

Let $l \in L$ and $e \in [T, T]$. $\text{cap}: L \times [T, T] \rightarrow \mathbb{N}$ is the function which describes the number of operations a module with module type l can execute in an interval e :

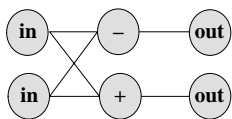
$$\text{cap}(l, e) = \left\lfloor \frac{|e| - d(l) + \text{dii}(l)}{\text{dii}(l)} \right\rfloor.$$

Consider the example DFG with the library given in figure 4. The optimal module set for time constraints of 2 or more cycle steps consists of 1 alu, and for the constraint of 1 cycle step it consists of 1 add and 1 sub module. The constraints in the last case can be formulated as:

$m(\text{add}, \{+, +\}) + m(\text{alu}, \{+, +\}) = I;$	constraints related to the additions
$m(\text{add}, \{+, +\}) \leq \mathbf{1} \times n(\text{add});$	
$m(\text{alu}, \{+, +\}) \leq \mathbf{1} \times n(\text{alu});$	

$m(\text{sub}, \{-, -\}) + m(\text{alu}, \{-, -\}) = I;$	constraints related to the subtractions
$m(\text{sub}, \{-, -\}) \leq \mathbf{1} \times n(\text{sub});$	
$m(\text{alu}, \{-, -\}) \leq \mathbf{1} \times n(\text{alu});$	

$m(\text{add}, \{+, -, +\}) + m(\text{alu}, \{+, -, +\}) = I;$	constraints related to both operation types
$m(\text{sub}, \{+, -, -\}) + m(\text{alu}, \{+, -, -\}) = I;$	
$m(\text{add}, \{+, -, +\}) \leq \mathbf{1} \times n(\text{add});$	
$m(\text{sub}, \{+, -, -\}) \leq \mathbf{1} \times n(\text{sub});$	
$m(\text{alu}, \{+, -, +\}) + m(\text{alu}, \{+, -, -\}) \leq \mathbf{1} \times n(\text{alu}).$	



Library:
 add (+), $d(\text{add}) = 1, c(\text{add}) = 1;$
 sub (-), $d(\text{sub}) = 1, c(\text{sub}) = 1;$
 alu (+, -), $d(\text{alu}) = 1, c(\text{alu}) = 1.5.$

Fig 4: Example DFG with library.

The values in italic represent numbers of operations in a had-interval. The bold **1**'s denote module capacities. If the time constraint would be 2 cycle steps, the bold **1**'s would have to be replaced by **2**'s. If several operation types can be mapped on the same module type, then extra constraints are needed to ensure that enough modules of that type are selected. The constraints on both operation types $\{+, -\}$ together are necessary: otherwise 1 alu would be selected instead of 1 add and 1 sub module. The constraints related to the distribution of operations can now be formalized:

CONSTRAINT SET 1A.

Let $ts \in P(\text{Type}), t \in ts$ and $\text{no}(ts, t)$ be the number of operations $v \in V, \tau(v) = t$ within the had-interval $e \in \varepsilon(ts)$. Then the following constraints must always be valid:

$$\forall_{ts \in P(\text{Type})} \forall_{t \in ts} : \sum_{l \in \mu(\{t\})} m(l, ts, t) = \text{no}(ts, t),$$

$$\forall_{ts \in P(\text{Type})} \forall_{l \in \mu(ts)} : \sum_{t \in ts | t \in \mu^{-1}(\{l\})} m(l, ts, t) \leq \text{cap}(l, e) \times n(l).$$

Remarks. The variables $n(l)$ are the only integer variables in the above and the following constraint sets! If for a set of operation types not all types are present in the had-interval, then the corresponding constraints can be omitted.

The execution interval of each operation is of course an upperbound for its execution time, a condition not yet taken care of. Consider for instance the had-interval of the + operation type of figure 2 with the following library:

add1 (+), $d(\text{add1}) = 1, c(\text{add1}) = 5;$
 add2 (+), $d(\text{add2}) = \text{dii}(\text{add2}) = 2, c(\text{add2}) = 2;$
 mult (*), $d(\text{mult}) = \text{dii}(\text{mult}) = 2; \text{decr}(-), d(\text{decr}) = 1.$

The constraints related to the + operation type are:

$$\begin{aligned} m(\text{add1}, \{+, +\}) + m(\text{add2}, \{+, +\}) &= 3; \\ m(\text{add1}, \{+, +\}) &\leq \mathbf{2} \times n(\text{add1}); \\ m(\text{add2}, \{+, +\}) &\leq \mathbf{1} \times n(\text{add2}). \end{aligned}$$

Consequently 3 add2 modules would be chosen. Figure 2 however shows $|EI(v_1)| = |EI(v_2)| = 1$, requiring mappings to add1. The respective constraint is: $m(\text{add1}, \{+, +\}) \geq 2$. In general terms we thus obtain the additional constraint set:

CONSTRAINT SET 1B.

Let $ts \in P(\text{Type}), t \in ts, i \in \mathbb{N}^+$ and let $\text{noo}(ts, t, i)$ be the number of operations $v \in V, \tau(v) = t, |EI(v)| \leq i$ within the had-interval $e \in \varepsilon(ts)$. Then the following constraints must be valid:

$$\forall_{ts \in P(\text{Type})} \forall_{t \in ts} \forall_{i \in [1, 2, \dots, |e|]} : \sum_{l \in \mu(\{t\}) | d(l) \leq i} m(l, ts, t) \geq \text{noo}(ts, t, i).$$

Remarks. Many of the constraints of set 1b are superfluous and can be omitted when:

- the left hand side is equal to the left hand side of another constraint, while the right hand side of the other constraint is equal or larger;
- the right hand sides are equal, while the left hand side of the other constraint is a subset.

4.1.2. Fixed operations

For operations which always occupy a module in a certain cycle step, new constraints can be derived. Again consider the example of figure 2 with the library stated above. Using constraint set 1a/b, an add1 and an add2 would be selected. As can be seen in figure 2, the operations v_1 and v_2 will always be scheduled in cycle 0. We also have seen that they must be mapped on add1 modules, so this leads to the new constraint: $n(\text{add1}) \geq 2$. Now 2 add1, 1 decr and 1 mult are selected, which constitute the optimal module set. This constraint set related to fixed operations can be formalized as:

CONSTRAINT SET 2.

Let $ts \in P(\text{Type})$, $i \in \mathbb{N}^+$ and $\text{nof}(ts, i)$ be the maximal number of operations $v \in V$, $\tau(v) \in ts$, $|EI(v)| \leq i$, which always occupy a module in the same cycle step. Then the following constraints must always be valid (many constraints can be omitted in the same way as constraints of set 1b):

$$\forall_{ts \in P(\text{Type})} \forall_{i \in [1, |T|]} : \sum_{l \in \mu(ts) \mid d(l) \leq i} n(l) \geq \text{nof}(ts, i).$$

4.1.3. Path delays

In the previous constraints it is not considered whether the minimal delay of each path in a DFG can be within the time constraint. The critical paths are those which could become the path with the largest minimal delay. Only these critical paths have to be considered in the respective constraints, and they can be determined efficiently by breadth first search. The corresponding constraints are as follows:

CONSTRAINT SET 3.

Let $k: L \times \text{Type} \rightarrow [0, \dots, 1]$ be a help function, C the set of critical paths, and $\text{nop}(c, t)$ the number of operations $v \in V$, $\tau(v) = t$ within the path $c \in C$. Then must be valid:

$$\forall_{c \in C} : \sum_{t \in \text{Type}} \sum_{l \in \mu(t)} (k(l, t) \times d(l) \times \text{nop}(c, t)) \leq |T|,$$

$$\forall_{t \in \text{Type}} : \sum_{l \in \mu(t)} k(l, t) = 1, \quad \forall_{t \in \text{Type}} \forall_{l \in \mu(t)} : k(l, t) \leq n(l).$$

4.2. Module selection review

If the scheduler does not succeed to make a correct schedule with the initial module selection, a review has to be made until a good selection has been found. The review can be done by altering the constants in the constraints. First the number of operations of a type can be increased above the actual number. Secondly, the number of available cycles in a distribution interval or a path can be decreased. Using such alterations, more and/or faster modules will be selected.

4.3. Resource constrained list scheduling

A list scheduler tries to select the best mappings for the operations in the ready list (the ready list contains the set of

nodes which can be scheduled in the current cycle step). [Gutb91] and [Rama91] try to estimate the influence of each possible mapping of operations, which can become very time consuming. The nodes in the ready list are therefore partitioned into equivalence classes in our approach [Grujij92]. Nodes are part of the same class if they have the same operation type, and if they have fully identical (but not necessarily the same) subgraphs succeeding them. The scheduler evaluates all the possible mappings of available modules on the equivalence classes (and not on all the operations in the ready list).

For that evaluation, the best set of operations within an equivalence class is selected, based on: the number of successors which are made available or partly available for scheduling and the number of common successors.

The evaluation of the mappings of modules on the equivalence classes is based on: the execution intervals and the number of the selected operations; the execution intervals and the number of operations which occur in the next ready list or which are made available for scheduling by the selected operations; the occupation of the modules in the current and next cycle step.

The evaluation can be extended, by not only regarding mappings in the current cycle step, but by evaluating also possible mappings in next cycle steps in a similar way. The depth level is therefore defined as the number of next cycle steps to be included in the evaluation.

5. Experiments and results

Results of our approach on two benchmarks are given in this section. The benchmarks are: WDELFF, the fifth order wave digital filter from [DeWi85] and FDCT, the fast discrete cosine transform from [Deny90]. FDCT contains in contrast to WDELFF a lot of parallelism which can be reduced significantly when the number of available cycle steps increases. The NEAT system has been implemented in C++ and the tests were run on a HP Apollo 750 workstation.

In Table 2 and 3 the results for the class 1 and 3 module libraries of Table 1 are given. The CPU times of the initial module selections were within 0.1 seconds and 0.6 seconds for respectively the class 1 and class 3 library. The initial selections for a class 1 library can be solved in polynomial time, just by ceiling the integer variables after running a LP solver. Only a few initial selections (less than 10%) did not lead directly to the optimal module set and required a module selection review. The resource constrained list scheduler always came up with the optimal solution (i.e. with the minimal numbers of cycle steps needed for the optimal module sets), and could always produce a schedule within a few seconds of CPU-time if no depth level was needed. There were only five class 3 cases and one class 1 case which needed a depth level between 1 and 4.

6. Conclusions

This paper presented a novel module selection and scheduling approach, which is able to select freely from a module library, in which several implementations (with different delays, DIIs and so on) for an operation can be available. The module selection is made before scheduling, to be able to deal efficiently and in a global manner with the design alternatives. The module selection is formulated as a MILP problem with only a few integer variables. The approach is therefore very efficient and we are able to find accurate constraints based on the concepts of the distribution of operations, fixed operations and path delays. The efficiency of solving this MILP problem depends upon the size of the library and not on the size of the DFG! Furthermore a new resource constrained scheduler has been developed, which is capable of handling an unrestricted module set efficiently.

Experiments showed the accuracy of the module selections and the good interaction between the module selection and the resource constrained scheduler, resulting in optimal solutions and very fast running times. A comparison between the results of different libraries makes clear that it is very advantageous to use unrestricted libraries, as they lead to designs with less module area.

References

- [Chen91] L.-G. Chen and L.-G. Jeng, "Optimal Module Set and Clock Cycle Selection for DSP Synthesis", Proc. ISCAS-91, pp. 2200-2203, 1991.
- [Deny90] P. Denyer, "SAGE - A User Directed Synthesis Tool", Proc. of the ASCIS Open Workshop on Synthesis Techniques for (lowly) Multiplexed Datapaths, Leuven, Belgium, August 1990.
- [DeWi85] P. DeWilde, E. Deprettere and R. Nouta, "Parallel and Pipelined VLSI Implementations of Signal Processing Algorithms", in S.Y. Kung, H.J. Whitehouse and T. Kailath, VLSI and Modern Signal Processing, Prentice Hall, pp. 258-264, 1985.
- [Gebo92] C.H. Gebotys, M.I. Elmasry, "Optimal VLSI Architectural Synthesis", Kluwer, 1992.
- [Gruij92] P.W.F. Gruijters, "Resource Constrained List Scheduling Using Unrestricted Libraries", Master Thesis, Eindhoven University of Technology, The Netherlands.
- [Gutb91] P. Gutberlet, H. Krämer and W. Rosenstiel, "CASCH - A Scheduling Algorithm for "High Level"- Synthesis", Proc. EDAC-91, pp. 311-315, 1991.
- [Ishi91] M. Ishikawa and G. DeMicheli, "A Module Selection Algorithm for High-Level Synthesis", Proc. ISCAS-91, pp. 1777-1780, 1991.

[Jain92] R. Jain, A.C. Parker and N. Park, "Predicting System-Level Area and Delay for Pipelined and Nonpipelined Designs", IEEE Trans. on CAD, vol. 11, no. 8, pp. 955-965, august 1992.

[McFa90] M.C. McFarland, A.C. Parker and R. Camposano, "The High-Level Synthesis of Digital Systems", Proc. of the IEEE, vol. 78, no. 2, pp. 301-318, 1990.

[Rama91] L. Ramachandran and D.D. Gajski, "An Algorithm for Component Selection in Performance Optimized Scheduling", Proc. ICCAD-1991, pp. 92-95, 1991.

Table 1: Class 1 and class 3 module library.

module type	area	delay in cycle steps	operations		
			*	+	-
mult	60	2	x		
alu1	9	1		x	x
sub1	8	1			x
add1	8	1		x	
alu2	6	2		x	x
sub2	5	2			x
add2	5	2		x	

The first two types are the only module types in the class 1 library, the other types also belong to the class 3 library.

Table 2: WDLF results.	resulting (optimal) module set							
	class 1		class 3					
	mult	alu1	mult	alu1	add1	sub1	alu2	sub2
[T]								
17	3	3	3	1	2			
18 to 20	2	2	2	1	1			
21 to 26	1	2	1		1		1	
27	1	2	1		1			1
28 to 53	1	1	1	1				
54 to	1	1	1				1	

Table 3: FDCT results.	resulting (optimal) module set							
	class 1		class 3					
	mult	alu1	mult	alu1	add1	sub1	alu2	sub2
[T]								
8 to 9	8	4	8		2	2		
10	5	4	5		2	2		
11	4	3	4	1	1	1		
12	4	3	4		1	1	1	
13	4	2	4		1	1		
14 to 17	3	2	3		1	1		
18	2	2	2	2				
19 to 25	2	2	2	1				1
26 to 33	2	1	2	1				
34 to 51	1	1	1	1				
52 to	1	1	1				1	