A module selection algorithm for high-level synthesis

Masaki Ishikawa* and Giovanni De Micheli

Center for Integrated Systems
Stanford University

Abstract

This paper presents a heuristic approach to the module selection problem in high-level synthesis. In contrast to the common assumption made by most high-level synthesis systems, which consider one available resource for each type of operation, we assume that several resources with different delays and areas are available in a functional-block library. The proposed algorithm solves the scheduling, resource sharing and module selection problems at the same time to achieve a circuit structure with near minimal area under a given overall latency constraint. Following the presentation of the algorithm, experimental results are reported.

1 Introduction

Most work in high-level synthesis addressed the *scheduling* and *module binding* problems [1]. When considered together, these problems are computationally hard and heuristic approaches have been proposed to solve them. It is a common assumption that one functional block can implement a given type of operation. For example, multiply operations would be bound to one specific hardware multiplier.

Module selection is the problem of choosing a particular functional unit from a library of components for each operation. The component library contains different alternative implementations for each resource type, that are characterized by different area and delay estimates. As an example, different hardware multipliers (e.g. parallel, iterative, ...), with different area and speed, can be chosen in the search for the best implementation. Module selection was originally considered by Leive [2], who proposed a method to decide how abstract structures could be implemented by hardware resources in a database.

Ideally, module selection, scheduling and binding should be solved concurrently. Because of their complexity, these problems have been tackled separately by using heuristic algorithms in existing high-level synthesis systems.

This paper proposes a new module selection algorithm for high-level synthesis. The algorithm combines module selection with scheduling and resource sharing. We model hardware behavior as a hypergraph [3]. We propose a heuristic algorithm, that has been successfully implemented in program *MSSR*. We also present some experimental results for standard benchmarks to show the effectiveness of the algorithm.

2 Problem formulation

We model hardware behavior as a *hypergraph H(V,E,S)*, where the *vertex set V*=$\{v\}$ represents the operations, the directed *edge set E*=$\{e\}$ (ordered vertex pairs) represents the dependencies and the *hyperedge set S*=$\{s\}$ (unordered subsets of V) represents the sharing of hardware among operations. We denote by v_0 and v_N the vertices corresponding to the first and last operation to be performed. We assume that the directed edges do not form cycles. The hyperedge set S forms a partition of V, i.e. we assume that operation-groups sharing a resource do not overlap and that each operation that is not shared is represented by an hyperedge as well, consisting only of the corresponding vertex. Hence:

$$\bigcup s = V \qquad s_i \bigcap s_j = \emptyset \ (i \neq j) \qquad |s| \geq 1.$$

A *resource sharing* configuration is defined by the set S and by a corresponding set of edges that represent the serialization of the operations bound to the shared resources. We indicate by S_0 the configuration with no shared resources. Note that each set s_i in S_0 includes one and only one $v_i \in V$ as element.

We assume to be given a *resource set R*=$\{r\}$ as a functional-block library. For each resource *r*, its delay-cost *delay(r)* and its area-cost *area(r)* are specified. The delay is a positive integer, corresponding to number of cycles needed to execute the operation. Each vertex and resource have a *type(r)*, such as *add, multiply* and *subtract*, representing the kind of operations. We assume that, for each resource *r*, $delay(r)$ is lower than the delay of any other resource with the same type and inferior area, i.e. if $delay(r_a) < delay(r_b)$ and $type(r_a) = type(r_b)$ then $area(r_a) > area(r_b)$. Vertices having the same *type* can share one resource with that *type* to execute the operation. A *trivial resource set* R_0 is a resource set which has only one resource for each *type*.

Given a resource sharing configuration S, a *module selection* is a mapping $\mu : S \rightarrow R$ associating one library element to each hyperedge. The overall area-cost is the sum of the area-cost of resources mapped to the hyperedges.

Given a resource sharing configuration S, a *schedule* is an integer labeling $\varphi : V \rightarrow Z^+$, such that $\varphi(v_j) \geq \varphi(v_i) + delay(r)$ $(r = \mu(s), v_i \in s)$ if there is an edge $(v_i, v_j) \in E$, where $\varphi(v_0) = 0$. We call $\lambda = \varphi(v_N)$ the *latency* of the hypergraph. We assume that the overall latency λ has a given upper bound $\overline{\lambda}$. We also define the earliest starting time $\epsilon(v)$ and the latest starting time $l(v)$ for each $v \in V$ as the minimum and maximum values of $\varphi(v)$ subject to the latency bound. Most work in high-level synthesis addressed the scheduling and resource sharing problems for *trivial* resource sets, that in this framework can be characterized as follows:

Given a hypergraph H(V,E,S_0), a trivial resource set R_0 and a maximum latency $\overline{\lambda}$, find a set S and a schedule $\varphi : V \rightarrow Z^+$, with $\varphi(v_N) \leq \overline{\lambda}$, that minimize the overall area.

Given a resource sharing configuration, the module selection problem can be stated as follows:

Given a hypergraph $H(V, E, S)$, a resource set R and a maximum latency $\overline{\lambda}$, find a mapping $\mu : S \rightarrow R$ and a schedule $\varphi : V \rightarrow Z^+$, with $\varphi(v_N) \leq \overline{\lambda}$, that minimize the overall area.

We consider in this paper a combination of the two problems, i.e. we consider the module selection problem together with scheduling and resource sharing:

*On leave from C&C Systems Research Laboratories, NEC Corporation, Kawasaki, Japan during the academic year of 1989-1990.

CH 3006-4/91/0000 – 1777 $1.00 © IEEE

Given a hypergraph $H(V, E, S_0)$, a resource set R and a maximum latency $\overline{\lambda}$, find a set S, a mapping $\mu : S \rightarrow R$ and a schedule $\varphi : V \rightarrow Z^+$, with $\varphi(v_N) \leq \overline{\lambda}$, that minimize the overall area.

3 The algorithm

Since the problem we want to solve is computationally hard, we use a heuristic approach. Our algorithm is based on iterative improvement. We repeat the following steps. First we compute a resource sharing configuration. For this configuration, we compute the best module selection, i.e. a module selection with the minimal overall area, such that the corresponding schedule does not violate the latency constraint. Then a new resource sharing configuration is tried by merging two hyperedges. These two steps are repeated until no improvement in the solution is found.

The algorithm consists of three procedures *INITRS*, *OPTSHR* and *OPTSLT* which are invoked and controlled by procedure *MAIN*.

- *INITRS* determines the initial hyperedge set. First we assume to be given a dependency graph with no resource sharing. For every *type* of vertices in the graph, the procedure finds iteratively a directed path in the graph that includes a maximal number of vertices with the *type* to make an initial hyperedge set S. The initial hyperedge set represents the initial resource sharing for the problem. The overall latency λ does not increase by the initial resource sharing because every vertex sharing one resource is already serialized by being on a directed path.

- *OPTSHR* chooses two hyperedges and merges them to make a new hyperedge. A distribution graph, as proposed by Paulin [4], is used to determine a schedule for vertices in the two hyperedges. To specify a new resource sharing represented by the new hyperedge, some directed edges may have to be added to the edge set *E*. For example let us consider a dependency graph shown in figure 1. The current hyperedge

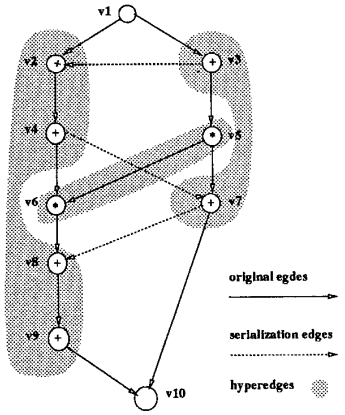

Figure 1: Merging two hyperedges.

set is $S = \{s_1, s_2, s_3\}$, where $s_1 = \{v_5, v_6\}, s_2 = \{v_2, v_4, v_8, v_9\}$ and $s_3 = \{v_3, v_7\}$. If two hyperedges, s_2 and s_3 are chosen to be merged and a schedule $\varphi(v_3) \leq \varphi(v_2) \leq \varphi(v_4) \leq \varphi(v_7) \leq \varphi(v_8) \leq \varphi(v_9)$ has been obtained by using a distribution graph, three edges $(v_3, v_2), (v_4, v_7)$ and (v_7, v_8) will be added to E to make a new edge set E_{new}.

- *OPTSLT* selects appropriate resources in the library to be mapped to each hyperedge. The procedure is based on the longest path algorithm. Starting from a mapping with the largest resource for each *type*, the procedure finds iteratively a smaller area resource to be mapped to a hyperedge. The selected resource is the largest among those that are smaller than the current one. Therefore the overall area decreases at the expense of increased latency. For each new mapping, the overall latency λ_{temp} is computed by using the longest path algorithm to check whether the given latency constraint is satisfied. If λ_{temp} is less than or equal to $\overline{\lambda}$, the new mapping is accepted as a better solution, otherwise the mapping is rejected and the procedure no longer tries to map a smaller resource to the hyperedge. When there are no smaller resources that can be mapped, the procedure returns a new mapping μ and $flag = 1$ if it has found a mapping with a smaller overall area. Otherwise it returns $flag = 0$.

- *MAIN* calls first *INITRS* and *OPTSLT*. Then it invokes iteratively procedures *OPTSHR* and *OPTSLT* until a smaller area solution under the overall latency constraint is found.

The algorithm can also solve the scheduling and resource sharing problems for a *trivial* resource sets and the module selection problem stated in the previous section, respectively. The complexity of procedures *INITRS*, *OPTSHR* and *OPTSLT* is $O(|V|(|V| + |E|)), O(\overline{\lambda}|V|^2)$ and $O(|R|(|V| + |E|))$, respectively.

4 Implementation and experimental results

The algorithm has been implemented in program *MSSR*, that is written in the C programming language. The program reads a dependency graph (*SIF*) generated by program *Hercules* [5], a functional block library and a maximum latency $\overline{\lambda}$. *MSSR* generates a schedule, a resource sharing configuration and a module selection that minimize the overall area.

Program *MSSR* was tested on a number of examples. Three sets of experimental results, based on benchmarks used in the literature, are presented here. For each example, two kinds of experimental results will be shown. The first result is obtained by using the same assumptions as in the original references, i.e. by using a *trivial* resource set as the library. Thus, the output of algorithm *OPTSHR* can be compared with those obtained by other systems. The second result is obtained by using a resource set which has several functional blocks, and it shows the performance of program *MSSR*.

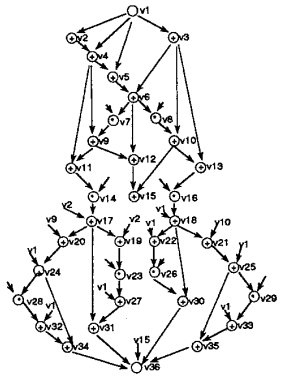

Figure 2: Dependency graph for the elliptic filter example.

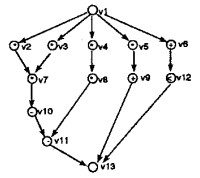

Figure 3: Dependency graph for the differential equation example.

A. Digital Elliptic Filter Example

The first example is taken from Kung's book on signal processing [6] and it was chosen as a benchmark for the 1988 High-Level Synthesis Workshop. This is a practical example which contains 26 add operations and 8 multiply operations. Figure 2 shows a dependency graph for this example.

Table 1 compares the results reported in the literature [4] [7] [8] with *MSSR*. We use a *trivial* resource set, which includes one adder and one

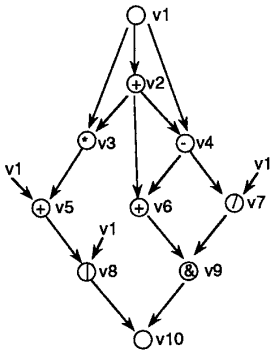

Figure 4: Dependency graph for the Facet example.

	FDS			FDLS			PBS			ALPS			MSSR		
$\overline{\lambda}$	17	18	21	17	18	21	17	18	21	17	18	21	17	18	21
#add	3	3	2	3	2	2	3	2	2	3	2	2	3	2	2
#mul	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1

Table 1: Digital elliptic filter with a trivial resource set.

$\overline{\lambda}$	λ	Area	Adders	Multipliers
14	14	576	*add1* × 4	*mpy1* × 2
15	15	304	*add1* × 3	*mpy1* × 1
16	16	293	*add1* × 2, *add2* × 1	*mpy1* × 1
18	18	288	*add1* × 2	*mpy1* × 1
30	27	272	*add1* × 1	*mpy1* × 1
60	60	176	*add1* × 3	*mpy2* × 4
70	70	144	*add1* × 1	*mpy2* × 4
100	93	80	*add1* × 1	*mpy2* × 2
160	156	37	*add2* × 1	*mpy2* × 1
300	288	36	*add3* × 2	*mpy2* × 1
450	448	34	*add3* × 1	*mpy2* × 1
1050	1040	12	*add3* × 2	*mpy3* × 4

Table 2: Digital elliptic filter with various resources.

Systems	FDS	FDLS	MSSR
$\overline{\lambda}$	4	4	4
# of ALU_a	2	2	2
# of ALU_b	2	2	2

Table 3: Differential equation with a trivial resource set.

multiplier. The delay of the adder and the multiplier are 1 and 2 respectively, to be compatible with the results achieved in other systems. The second row of Table 1 indicates the given maximum latencies $\overline{\lambda}$. The third and the fourth row show the number of resources used to synthesize the circuit. *MSSR* solves the problem with the minimal number of adders and multipliers for each $\overline{\lambda}$.

Table 2 shows an experimental result obtained by using several functional blocks. We assume the following resource set:

adder:
add1: delay(*add1*) = 1, area(*add1*) = 16
add2: delay(*add2*) = 4, area(*add2*) = 5
add3: delay(*add3*) = 16, area(*add3*) = 2

multiplier:
mpy1: delay(*mpy1*) = 1, area(*mpy1*) = 256
mpy2: delay(*mpy2*) = 16, area(*mpy2*) = 32
mpy3: delay(*mpy3*) = 256, area(*mpy3*) = 2

We assume a 16 bits word length for each operation. The different operation types correspond to a parallel, serial/parallel and a serial implementation respectively. The area of each resource is estimated by the number of full adders and flip–flops used in the resource.

Program *MSSR* was executed with several values of $\overline{\lambda}$. The module selection results are shown in Table 2. The first column of Table 2 shows $\overline{\lambda}$. The second and the third column show the computed overall time λ and the overall area, respectively. The last two columns show the resources used.

B. Differential Equation Example

This example is taken from a paper describing *HAL* system [4]. The example is a circuit to solve a differential equation and it includes six multiply operations, two add operations, two subtract operations and one compare operation in the dependency graph, as shown in figure 3.

Table 3 shows a result obtained by *MSSR* with a *trivial* resource set and the results achieved by *HAL*. We assume two ALUs, ALU_a and ALU_b, as functional blocks. ALU_a can execute add, subtract and compare operations, while ALU_b can execute a multiply operation. Both resources have delay = 1.

Table 4 shows the results obtained by using several functional blocks. We assume the following resource set:

ALU_a(add, sub and compare):
alu_a1: delay(*alu_a1*) = 1, area(*alu_a1*) = 24
alu_a2: delay(*alu_a2*) = 4, area(*alu_a2*) = 7
alu_a3: delay(*alu_a3*) = 16, area(*alu_a3*) = 3

ALU_b(multiply):
alu_b1: delay(*alu_b1*) = 1, area(*alu_b1*) = 256
alu_b2: delay(*alu_b2*) = 16, area(*alu_b2*) = 32
alu_b3: delay(*alu_b3*) = 256, area(*alu_b3*) = 2

C. Facet example

This example is taken from [9]. It includes three add operations, one subtract operation, one multiply operation, one divide operation, one logical *and* (&) and one logical *or* (|) operation. Figure 4 shows a dependency graph for the example.

Table 5 shows the experimental results with a *trivial* resource set as a library. It compares our result with those obtained by other synthesis systems [9] [10] [11]. The library has three ALUs, *ALU1*, *ALU2* and *ALU3* . Each of the ALU resources has delay = 1.

Table 6 shows the results obtained by using several functional blocks. We assume the following three *types* of ALUs.

ALU_a(add and sub):
alu_a1: delay(*alu_a1*) = 1, area(*alu_a1*) = 20
alu_a2: delay(*alu_a2*) = 4, area(*alu_a2*) = 6
alu_a3: delay(*alu_a3*) = 16, area(*alu_a3*) = 2

ALU_b(multiply and divide):
alu_b1: delay(*alu_b1*) = 1, area(*alu_b1*) = 384
alu_b2: delay(*alu_b2*) = 16, area(*alu_b2*) = 48
alu_b3: delay(*alu_b3*) = 256, area(*alu_b3*) = 3

$\overline{\lambda}$	λ	Area	ALU_a	ALU_b
4	4	560	*alu_a1* × 2	*alu_b1* × 2
5	5	536	*alu_a1* × 1	*alu_b1* × 2
7	7	280	*alu_a1* × 1	*alu_b1* × 1
12	12	270	*alu_a2* × 2	*alu_b1* × 1
20	20	263	*alu_a2* × 1	*alu_b1* × 1
40	40	110	*alu_a2* × 2	*alu_b2* × 3
60	60	71	*alu_a2* × 1	*alu_b2* × 2
100	100	39	*alu_a2* × 1	*alu_b2* × 1
520	520	15	*alu_a2* × 1	*alu_b3* × 4

Table 4: Differential equation with various resources.

Systems	Facet	Splicer	ADPS	MSSR
λ	4	4	4	4
ALU1	+, *, |	+ *, |	+, |	+, |
ALU2	+, −, &	+, −, &	+, −, &	+, −, &
ALU3	/	/	*, /	*, /

Table 5: Facet example with a trivial resource set.

$\bar{\lambda}$	λ	Area	ALU_a	ALU_b	ALU_c
4	4	428	*alu_a1* × 2	*alu_b1* × 1	*alu_c1* × 2
5	5	406	*alu_a1* × 1	*alu_b1* × 1	*alu_c1* × 1
7	7	405	*alu_a1* × 1	*alu_b1* × 1	*alu_c2* × 1
14	14	397	*alu_a2* × 2	*alu_b1* × 1	*alu_c2* × 1
18	18	391	*alu_a2* × 1	*alu_b1* × 1	*alu_c2* × 1
19	19	120	*alu_a1* × 1	*alu_b2* × 2	*alu_c1* × 2
20	20	118	*alu_a1* × 1	*alu_b2* × 2	*alu_c2* × 2
22	22	117	*alu_a1* × 1	*alu_b2* × 2	*alu_c2* × 1
28	28	103	*alu_a1* × 2	*alu_b2* × 2	*alu_c2* × 1
34	34	70	*alu_a1* × 1	*alu_b2* × 1	*alu_c1* × 1
38	38	55	*alu_a2* × 1	*alu_b2* × 1	*alu_c2* × 1
52	52	53	*alu_a3* × 2	*alu_b2* × 1	*alu_c2* × 1
66	66	51	*alu_a3* × 1	*alu_b2* × 1	*alu_c2* × 1
259	259	30	*alu_a1* × 1	*alu_b3* × 2	*alu_c1* × 2
260	260	28	*alu_a1* × 1	*alu_b3* × 2	*alu_c2* × 2
262	262	27	*alu_a1* × 1	*alu_b3* × 2	*alu_c2* × 1
268	268	13	*alu_a2* × 1	*alu_b3* × 2	*alu_c2* × 1
292	292	9	*alu_a3* × 1	*alu_b3* × 2	*alu_c2* × 1
529	529	7	*alu_a3* × 1	*alu_b3* × 1	*alu_c1* × 1
530	530	6	*alu_a3* × 1	*alu_b3* × 1	*alu_c2* × 1

Table 6: Facet example with various resources.

ALU_c(& and |):
alu_c1: delay(*alu_c1*) = 1, area(*alu_c1*) = 2
alu_c2: delay(*alu_c2*) = 2, area(*alu_c2*) = 1

Table 7 shows average CPU run times to obtain each result for the examples. CPU run times are reported in seconds while running on DecStation 3100 with 16MBytes of memory. Figure 5 shows delay/area curves representing experimental results shown in Table 2, Table 4 and Table 6.

5 Summary

We have formulated the problem of module selection with scheduling and resource sharing by using a hypergraph model. We have proposed a heuristic, polynomial-time algorithm that has been implemented in program *MSSR*. Experimental results show that *MSSR* can perform optimal scheduling and resource sharing on several recent benchmarks with a *trivial* resource set. Moreover, the program can provide a spectrum of solutions with an efficient area/delay trade-off, when different functional blocks for each operation are given as a resource set. Further research will be devoted to investigate an extension of the algorithm to incorporate interconnection and storage costs.

	Example A		Example B		Example C	
	trivial	various	trivial	various	trivial	various
CPU(s)	7.2	9.5	< 0.1	0.5	< 0.1	0.1

Table 7: CPU run times for the examples.

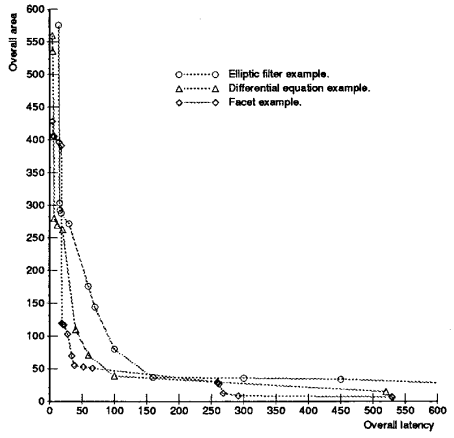

Figure 5: Area/delay trade-off for the examples.

6 Acknowledgments

David Ku implemented the program *Hercules* used for behavioral synthesis. Rajesh Gupta provided the hardware descriptions of the examples in *Hardware C*. The authors acknowledge the support from NSF, under a PYI award and grant No. MIP 8719546.

References

[1] M. C. McFarland, A. C. Parker and R. Camposano, "The high-level synthesis of digital systems," *Proc. of the IEEE*, vol.78, pp. 301-318, Feb. 1990.

[2] G. W. Leive and D. E. Thomas, "A technology relative logic synthesis and module selection system," in *Proc. 18th Design Automat. Conf.* 1981, pp. 479-485.

[3] C. Berge, *Graphs and Hypergraphs*. North-Holland. 1973.

[4] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Trans. Computer-Aided Design*, vol. CAD-8, pp. 661-679, June 1989.

[5] G. De Micheli, D. Ku, F. Mailhot and T. Truong, "The Olympus Synthesis System," *IEEE Design and Test of Computers*, pp. 37-53, October 1990.

[6] S. Y. Kung, H. J. Whitehouse and T. Kailath, *VLSI and Modern Signal Processing*. Englewood Cliffs, NJ: Prentice Hall. 1985, pp. 258-264.

[7] R. Potasman, J. Lis, A. Nicolau and D. Gajski, "Percolation Based Synthesis," in *Proc. 27th Design Automat. Conf.* 1990, pp. 444-449.

[8] C. T. Hwang, Y. C. Hsu and Y. L. Lin, "Optimum and heuristic data path scheduling under resource constraints," in *Proc. 27th Design Automat. Conf.* 1990, pp. 65-70.

[9] C. J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 379-385, July 1986.

[10] B. M. Pangrle, "Splicer: A heuristic approach to connectivity binding," in *Proc. 25th Design Automat. Conf.* 1988, pp. 536-541.

[11] C. A. Papachristou and H. Konuk, "A Linear program driven scheduling and allocation method following by a interconnect optimization algorithm," in *Proc. 27th Design Automat. Conf.* 1990, pp. 77-83.