Efficient Algorithms for Finding Highly Acceptable Designs Based on Module-Utility Selections *

Chantana Chantrapornchai Edwin H.-M. Sha Xiaobo (Sharon) Hu
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
email: {cchantra,esha,shu}@cse.nd.edu
Phone number: (219) 631-8803
Fax number: (219) 631-9260

Abstract

In this paper, we present an iterative framework to solve module selection problem under resource, latency, and power constraints. The framework associates a utility measure with each module. This measurement reflects the usefulness of the module for a given a design goal. Using modules with high utility values will result in superior designs. We propose a heuristic which iteratively perturbs module utility values until they lead to good module selections. Our experiments show that the module selections formed by combinations of modules with high utility values are superior solutions. Further, by keeping modules with high utility values, the module exploration space can drastically be reduced.

1 Introduction

Module selection in high-level synthesis is a complex problem due to its interaction with the scheduling and binding processes. Most existing high-level synthesis tools assume there is only one resource type of each functional unit during the scheduling phase. However, in some cases, functional units with the same functionality may use different modules to implement. This assumption can lead to an incomplete characterization of design space. Therefore, to ensure the fully design space exploration, module selection process needs to be integrated into the high-level synthesis tools.

In this paper, we present an efficient module selection framework which takes the effect of scheduling into account for a constrained environment. The success of the approach lies on the use of the *module utility* and *inclusion scheduling*. We called the approach *utility selection*. Moreover, we allow the constraint to be modeled as a fuzzy constraint, namely *acceptability function*, which takes arguments as design attributes. In return, the acceptability value describes how good the given design characteristics are. Such a proposed model can express a design objective, tradeoff between conflicting design criteria, as well as hard (fixed) or soft (fuzzy) constraint boundary.

Many researchers have been studying scheduling, binding and module selection problems. Torbey and Knight proposed a method which used a genetic algorithm to solve the scheduling and storage optimization problems [7]. However, their method excludes the fact that functional units with the same function may be implemented by different modules. Ahmad et.al. also used a genetic algorithm to solve the integrated problem of scheduling, binding, and module selection [1]. Their formulation, nonetheless, does not include fuzzy constraints. Ishikawa and De Micheli proposed a heuristic to find a module selection under latency and area constraints [4]. Their approach is neither applicable under fuzzy environment nor easily expandable to consider other criteria. Several works presented heuristics to completely characterize the design space [6, 3]. However, for a large application and module set, exploring all designs are still an expensive approach.

*This work was supported in part the NSF under grant number MIP 95-01006, MIP-9612298 and MIP-9701416.

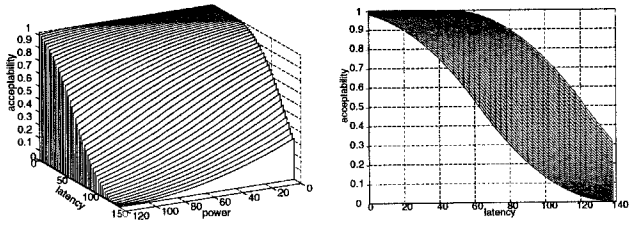
2 Models and Problem Descriptions

Operations and their dependencies in an application are modeled by a vertex-weighted directed acyclic graph, called a *Data Flow Graph*, $G = (V, E, \beta)$, where each vertex in the vertex set V corresponds to an operation and E is the set of edges representing data flow between two vertices. Function β defines the type of operation for node $v \in V$. Operations in a data flow graph can be mapped to different functional units which in turn can be implemented by different modules. Such a system must also satisfy certain design constraints, for instance, power and cost limitation. These specifications are characterized by a tuple $S = (N, \mathcal{F}, \mathcal{M}, A, \mathcal{Q})$, where N is the number of functional units allowed in the system, $\mathcal{F} = \{f_i : \forall i \in [1, N]\}$ is the set of functional units allowed in the system, e.g., {add, mul}. $\mathcal{M} = \{M_{f_i} | \forall f \in \mathcal{F}, \forall i \in [1, N]\}$, where each M_{f_i} contains a set of eligible modules for functional unit f_i, e.g., M_{f_1} = {ripple_adder, carry-look-ahead_adder}. A is a function mapping from $M_{f_i} \in \mathcal{M}$ to a set of tuples $(a_1, \ldots, a_k)$, where a_1 to a_k represent attributes of a particular module. In this paper, we are interested in synthesizing a system under latency and power constraints. Hence, $A(m) = (a_1, a_2)$ where a_1 refers to the latency attribute of module m while a_2 refers to the power consumption of module m. In this paper, we are interested in the *average* power consumption per time unit, using the formula in

0-7695-0104-4/99 $10.00 © 1999 IEEE

[5]: $P = \frac{\sum_k n_k p_k t_k}{T}$. T is the schedule length, p_k and t_k are the power consumption as well as execution time of functional unit f_k, and n_k is the number of operations executed by functional unit f_k. Finally, $\mathcal{Q}$ is a function that defines the degree of a system being acceptable for different system attributes. If $\mathcal{Q}(a_1, \ldots, a_k) = 0$ the corresponding design is totally unacceptable while $\mathcal{Q}(a_1, \ldots, a_k) = 1$, the corresponding design is definitely acceptable.

Using a function $\mathcal{Q}$ to define the acceptability of a system is a very powerful model. It can not only define certain constraints but also express design goals. In Figure 1(a), an example of z-curve shape fuzzy constraints is shown. The boundary is (latency $t = 150$, power $p = 120$) while the objective is to optimize the weighted sum of latency and power $2t + p$ (tradeoff ratio 2:1). Figure 1(b) presents the projection of the function in Figure 1(a) into a latency and acceptability plane. An inner z-curve (tighter latency) corresponds to a looser power constraint.



(a) z acceptability (b) Latency-Acceptability

Figure 1. Example of of acceptability functions

In this paper, the combined scheduling/binding and module selection problem we intend to solve can be formulated as follows: *Given a specification containing* $S = (N, \mathcal{F}, \mathcal{M}, A, \mathcal{Q})$, *and* $G = (V, E, \beta)$, *select modules* $m \in M_{f_i}$ *for a functional unit* f_i, $\forall f_i$, *based on the resulting module utility values while maximizing the acceptability degree of the solutions executing graph* G.

3 Utility Selection Framework

We use the concept of *utility values* to compute the utility selection. Each module is associated with a utility value, which represents the usefulness of a module. The module may be present in good designs which optimize a certain goal and/or bad designs that do not satisfy the design goal. We allow the utility value of a module to be any real number between 0 and 1 to represent this ambiguity. Note that the design using those modules with utility value of 1 should be of the highest quality.

The operations of the framework at a high level is straightforward. First, a designer give initial utility values. Then, the framework improves them until they lead to good module selections. Figure 2 depicts the utility selection framework in details. It consists of two main steps, scheduling and utility value improvement. The first phase, inclusion scheduling, takes a data flow graph, the number of functional units required in a target system, and a module set as well as their associated utility values as inputs and constructs a general schedule. Rather than creating all possi-

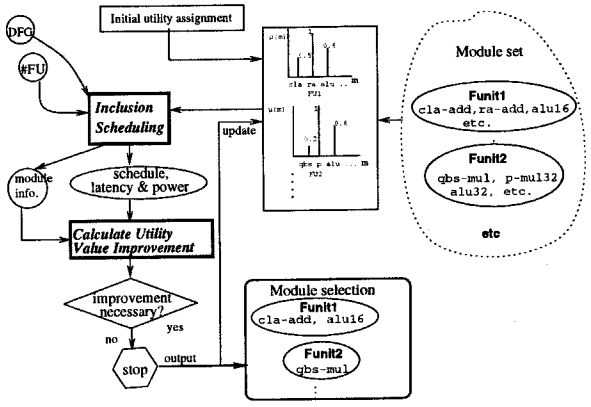


Figure 2. Utility selection framework

ble schedules, inclusion scheduling creates only one schedule for the entire module combinations for efficiency. More importantly, it produces varying latencies and powers which are close approximation of the latencies and powers generated by the schedules of all module combinations [2]. Such informations are useful for future assessment of modules' usefulness.

Particularly, while calculating a schedule, latency, power as well as the corresponding module usages are recorded. These data are then used as inputs to the second phase: the utility value improvement. In this step, the utility value of each module is adjusted. For a given module, the acceptability of every latency and power pair that the module contributes to is analyzed. Intuitively, if a module causes a lot of unacceptable latency and power values, the utility of the module should be decreased. On the other hand, if a module contributes to a lot of high-acceptability latency and power values, the module's utility value should be increased. The statistics of a module usage for each latency and power value are used as a scaling factor to the acceptability value for the module for signifying the module's contribution. Based on this idea, we have developed a heuristic to compute the relative adjustment of a utility value. The adjustment is then applied to update the previous utility value. The 2-step process is repeated until the adjustment values converge to zero. The experimental results show that the average number of iterations is approximately eleven.

Inclusion Scheduling

In order to construct an inclusion schedule based on a utility assignment, we borrow some techniques form the fuzzy theory [8]. In particular, we model modules and their respective utility values as a fuzzy set with respect to the corresponding functional unit. For functional unit f, and its eligible module set M_f, $\mu_f(m) \in [0, 1], \forall m \in M_f$, describes a utility value of module m. This

concept implicitly shows that a functional unit has fuzzy execution times and powers.

Inclusion scheduling is a scheduling method which takes into consideration of fuzzy characteristics, e.g., fuzzy set of varying latency (power) values associated with each functional unit. The output schedule, in turn, also consists of fuzzy attributes. The actual steps in inclusion scheduling is given in Algorithm 3.1. In a nutshell, inclusion scheduling simply replaces the computation of accumulated execution times in a traditional scheduling algorithm by the fuzzy arithmetic-based computation: $\mu_{A*B}(z) = \bigvee_{z=x*y}(\mu_A(x) \wedge \mu_B(y))$, where $\vee$ and $\wedge$ denote max and min operations respectively, $*$ is an arithmetic operation, e.g., maximum, addition, etc. In our case, z is a tuple of latency and power attributes. Hence, fuzzy arithmetics is used to compute possible latencies and powers from the given functional specification. Then, latencies and powers of different schedules are compared to select a functional unit for scheduling an operation.

In Algorithm 3.1, routine *Eval_Schedule* (Line 9) is where fuzzy arithmetic is applied. It computes the inclusion schedule and evaluate whether the schedule is better than a previous one. To compute an inclusion schedule, an array, whose element contains a tuple (latency,enery,util), is constructed to represent a fuzzy set of time and energy. This array stores possible latency and power of a current schedule and is updated while a node is being scheduled. Using this model, one can easily expand the design criteria by modifying the array structure and the fuzzy calculation part. At the end of the routine, two input fuzzy sets associated with the schedule *good_S* and *temp_S* are compared and the better one of the two is chosen.

Algorithm 3.1 (Inclusion scheduling)
Input: G = (V,E, β) *and*
specification Spec = (N, F, $\mathcal{M}$, A, $\mathcal{Q}$)
Output: *A schedule* S, *latency and power*

```
1  Q = vertices in G with no incoming edges
2  while Q ≠ empty do
3      Q = prioritized(Q)
4      u = dequeue(Q); mark u scheduled
5      good_S = NULL;
6      foreach f ∈ {f_j : where f_j is able to perform β(u)} do
7        // find a place to schedule u
8        temp_S = assign(S, u, f)
9        if Eval_Schedule(good_S, temp_S, G, Spec)
10         then good_S = temp_S
11     S = good_S
12     foreach v : (u, v) ∈ E do
13       indegree(v) = indegree(v) − 1
14       if indegree(v) = 0 then enqueue(Q, v)
16 return(S)
```

Utility Adjustment Heuristic

Recall that the utility values of modules should reflect the usefulness of the modules towards a design goal. In our heuristic, we compute the positive contribution $\sigma_+(f, m)$ and negative contribution $\sigma_-(f, m)$ of module m if it were used to implement functional unit f, and use them to compute the relative adjustment value for each iteration. The adjustment value is computed using the following:

$$\mathsf{adj}_f(m) = \frac{\sigma_+(f,m) - \sigma_-(f,m)}{\sigma_+(f,m) + \sigma_-(f,m)} \text{ where}$$

$$\sigma_+(f,m) = \sum_{\forall(t,p)\ \text{s.t.}\ \mu(t,p)=\mu_f(m)} \mathsf{freq}_{t,p}(f,m)\,\mu_{acc}(t,p)$$

$$\sigma_-(f,m) = \sum_{\forall(t,p)\ \text{s.t.}\ \mu(t,p)=\mu_f(m)} \mathsf{freq}_{t,p}(f,m)\,(1-\mu_{acc}(t,p))$$

The key information here is $\mathsf{freq}_{t,p}(f, m)$, the number of module references for each latency and power value pair, which will lead to appropriate module utility assignments. This value is obtained by modifying inclusion scheduling to also tally the module contributing to each pair.

Then, we apply $\mathsf{adj}_f(m)$ value in the following manner: if $\mathsf{adj}_f(m)$ equals 1, we double the value of $\mu_f(m)$ and if $\mathsf{adj}_f(m)$ equals -1, $\mu_f(m)$ is reduced by half. If $\mathsf{adj}_f(m)$ is between $(-1, 0]$, the change of $\mu_f(m)$ is proportional to half of $\mu_f(m)$ and if $\mathsf{adj}_f(m)$ is between $(0, 1)$, the change of $\mu_f(m)$ is proportional to $\mu_f(m)$. After the adjustment for all modules is made, $\mu_f(m)$ are normalized with respect to the highest one, i.e., $\text{norm}(\mu_f(m)) = \frac{\mu_f(m)}{\max_m \mu_f(m)}, \forall m \in M_f$. If $\mu_f(m)$ is the same as $\mu_f(m)$ from the previous iteration for every m, the adjustment is no longer needed.

4 Experimental Results

We performed several experiments on well-known benchmarks. Table 2 presents some of the results of the experiments from elliptic filter and discrete cosine transform. The module set used for these tests are shown in Figure 1.

In Table 2, Column "Spec" displays a specification of a target system. The linear acceptability functions used in these tests are described in Column "Acceptability $\mathcal{Q}$". Fields "latency" and "power" display two vital points (x_0, x_1) in which any acceptability is in between [0,1]. Column "$w_1 : w_2$" displays a tradeoff ratio between latency and power of each respectively. Column "Selected modules" of Table 2 shows the results generated by our method.

Modules	Time	Power
a0	5	60
a1	10	38
a2	20	23
a3	35	20
a4	40	10
a5	70	5

Modules	Time	Power
m0	100	296
m1	160	84
m2	170	70
m3	300	55
m4	640	29
m5	770	20

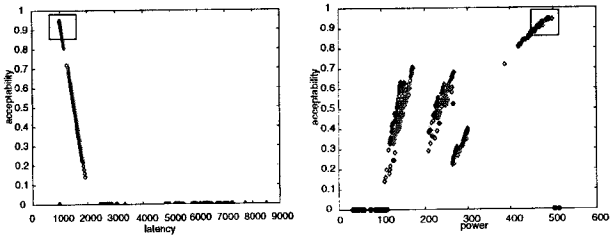
Table 1. Example module set

Take the discrete cosine transform benchmark (the second last rows if Table 2) as an example. Since the goal is to optimize 5t+p, we attempt to minimize latency at the expense of increasing power. Figure 3 characterizes enumerated solutions (generated by computing the schedules for all possible module combinations) in 2-dimensional latency and acceptability plane as well as power and acceptability plane. The points in the square box area correspond to the top rank solutions whose latencies are between [980, 1160]

Ben.	Spec.	Acceptability $\mathcal{Q}$			Selected modules	
		latency	power	$w_1 : w_2$	adder	mult
elf	2a 2m	(1300,1700)	(75,150)	5:1(l)	(a1 a2) (a2)	(m1 m2) (m1 m2)
elf	2a 2m	(1300,1700)	(75,150)	1:3(l)	(a2 a3 a4) (a2)	(m1) (m1)
dct	2a 2m	(1200,2600)	(75,150)	5:1(l)	(a2) (a2)	(m1 m2) (m1 m2)
dct	2a 2m	(1200,2600)	(75,150)	1:3(l)	(a2 a3 a4 a5) (a0)	(m1) (m2)
dct	3a 2m	(1000,2000)	(200,500)	5:1(l)	(a2) (a2) (a1 a2)	(m0) (m0)
dct	3a 2m	(1000,2000)	(200,500)	1:3(l)	(a1 a2) (a1 a2) (a1 a2)	(m1 m2) (m1 m2)

Table 2. Selected experimental results

and powers are between [418, 498]. Both of our selections of Table 2, (a2,a2,a2,m0,m0) and (a2,a2,a1,m0,m0), resulting in the latency and power (980, 498) and (1010,481) respectively, also lies in this region.



(a) Latency and acc. (b) Power and acc.

Figure 3. Enumerated solutions for the dct experiment in the second last row in Table 2

Table 3(a) summarizes the ratio between the number of enumerated solutions and average size of the derived elite set (formed by combinations of the selected modules) for each benchmark. Examining Column "% Reduction", one can see that our proposed approach is able to reduce the search space dramatically.

Table 3(b) shows the average quality of selections found by utility selection. We divide the solutions into 7 groups according to the acceptabilities of their corresponding latency and power values. Columns "Enum" and "USel" shows the average percentage of module selections per rank for both enumerated set and the elite set obtained by our algorithm. Though on average, most of module selections from the enumerated set can result in schedules with low acceptability degrees, our algorithm is able to capture good module selections. This also can be done within a reasonable amount of time. In the worst case example, discrete cosine transform with 5 functional units, the average running time of utility selection is about 800 seconds while it takes approximately 10800 seconds to generate 7776 schedules for all module combinations. Therefore, utility selection process takes only 7.4% of the running time of the enumerated approach. From all the tables, one can see that utility selection efficiently produces high-quality module selections.

Ben.	Spec	USel:Enum	%Reduction
test1	1a 2m	4 : 48	91%
test2	1a 2m	1.5 : 294	99%
deq	1a 2m	9 : 294	97%
fj-wdf	2a 1m	2.5 : 252	99%
fj-wdf	2a 2m	3.5 : 1764	99%
vtf	1a 2m	4.5 : 2058	99%
elf	2a 1m	5 : 216	97%
elf	2a 2m	4 : 1296	99%
dct	2a 2m	3.25 : 1296	99%
dct	3a 2m	11 : 7776	99%

(a)

Rank#	Enum	USel
1st	25%	79%
2nd	7%	11%
3rd	7%	1%
4th	6%	5%
5th	8%	1%
6th	3%	2%
7th	44%	1%

(b)

Table 3. (a) # elite set and #enumerated set ratio (b) Average distribution per rank

5 Conclusion

We have presented a module selection framework that takes into account of scheduling effect as well as resource, latency, and power constraints. This approach uses of the utility measure to model the degree of usefulness of a module. The scheduling and binding method called *inclusion scheduling* is used exclusively as a basis for deriving fuzzy latency and power values, approximating latency and power enumerated exhaustively, in order to improve module utility to reflect real module goodness. Experiments show that module utilities are good pointers to module selections. By using module utility, designers also have alternative in selecting initial designs. Our current approach can be integrated in an iterative design process varying the number of functional units for complete design exploration.

References

[1] I. Ahmad, et. al. Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis. *IEE Proc.-Comput. Digit. Tech.*, 142:65–71, Jan 1995.

[2] C. Chantrapornchai, E. H. Sha, and X. S. Hu. Efficient scheduling for imprecise timing based on fuzzy theory. In *Proc. MWCAS*, 1998.

[3] S. Chaudhuri, et. al. An exact methodology for scheduling in 3D design space. In *Proc. ISSS*, pp. 78–83, 1995.

[4] M. Ishikawa and G. De Micheli. A module selection algorithm for high-level synthesis. In *Proc. ISCAS*, pp. 1777–1780, 1991.

[5] R. Martin and J. P. Knight. Power-profiler: Optimizing ASICs power consumption at the behavioral level. In *Proc. DAC*, pp. 42–47, 1997.

[6] Z. X. Shen and C. C. Jong. Exploring module selection space for architectural synthesis of low power designs. In *Proc. ISCAS*, pp. 1532–35, 1997.

[7] E. Torbey and J. Knight. Performing scheduling and storage optimization simultaneously using genetic algorithms. In *Proc. MWCAS*, 1998.

[8] L. A. Zadeh. Fuzzy Logic. *Computer*, 1:83–93, 1988.