Distributed Design-Space Exploration for High-Level Synthesis Systems

Rajiv Dutta, Jayanta Roy and Ranga Vemuri
Laboratory for Digital Design Environments
Department of Electrical and Computer Engineering
University of Cincinnati, ML 30
Cincinnati, Ohio 45221-0030

Abstract
A parallel algorithm for design-space exploration and trade-off analysis is presented. Coarse-grained parallelism is introduced by generating multiple module bags and performing scheduling and performance analysis of the data flow graph for each module bag in parallel. This algorithm is implemented on a multiple processor machine as part of a distributed high-level synthesis system. Experimental results show reduction in search time, improvement in design quality, and close-to-linear speedup.

1 Introduction
High-level synthesis is the process of generating register-level designs given a behavioral specification, performance goals usually in terms of area and clock period, and a register-level module library. The result of high-level synthesis is a register-level data path which contains modules (registers, functional units etc.) selected from the module library and a finite-state controller to control the sequencing of register transfers in the data path. The high-level synthesis process is usually divided into three tasks: scheduling, allocation and binding. The behavioral specification is typically represented as a data dependency graph which indicates the potential parallelism among the various abstract operations. During the scheduling phase, the abstract operations in this graph are assigned to control steps. Scheduling algorithms take one or more of the following constraints into account: (1) Resource constraint - only the modules available in a module bag can be used; (2) Clock-period constraint - the maximum module delay in any control step should not exceed the user-specified limit on the clock period; (3) Area constraint - the total area occupied by the modules used in the schedule should not violate the user-specified area constraint; (4) Schedule length constraint: the length of the longest path in the scheduled graph should not exceed the user-specified limit. Most scheduling algorithms take resource constraint and clock-period constraint into account. Some scheduling algorithms produce the shortest possible schedule that meets the stated area constraint, whereas others attempt to produce the best area-efficient schedule while meeting the given schedule-length constraint. [7, 10, 11, 9]

Scheduling phase has a profound effect on the overall performance (both area and speed) of the synthesized design. It is important to generate various schedules representing various regions of the design-space and select the best possible schedule. Performance estimation procedures can be used to estimate the area/time characteristics of the designs which would result from a scheduled data flow graph; the best schedule can then be chosen either by the user or by the synthesis system based on a user-supplied cost function. Given the resource, clock-period, area and/or schedule length constraints, very good scheduling algorithms exist to determine near-optimal schedules [12, 2, 1]. Of the four types of constraints, usually the clock-period, area and schedule-length constraints, if any, come from the user. The resource constraint, however, must be generated by the synthesis system based on the module library provided. Clearly, different module bags lead to different schedules which in turn lead to different designs. Given a data flow graph and a module library, a vast number of module bags can be formulated to drive the scheduling algorithm. Arriving at a good schedule for each module bag, estimating the design performance for each schedule, and finally selecting the best among these schedules can be very expensive in computing time. Fortunately, since the schedules under different module bags can be arrived at independently of each other, the computing power of multiple processor systems can be effectively used to reduce this time.

This paper presents an algorithm for module bag selection and parallel design-space exploration, based on this simple idea shown as Algorithm 1. We admit parallelism at the scheduling algorithm level. Each processor in the multiple processor system executes the same scheduling algorithm. Each scheduler has a copy of the same data flow graph and the same area and clock-period constraints. What is different among the schedulers is the resource constraint; each scheduler is driven by a different module bag selected from a pool of valid module bags created a priori. Each scheduler attempts to generate the shortest possible schedule. The scheduling phase in each processor is followed by a performance estimation phase where the area and speed of the design that would result with this schedule are estimated. The performance results are then gathered and presented (to the user or the synthesis system) in the form of tradeoff curves where different tradeoff points correspond to different module bags. The best tradeoff point (and module bag) may then be selected (by the user or the synthesis system) and the corresponding schedule may be passed on to the register allocation and connection optimization phases to complete the design synthesis process.

The number of valid module bags generated (Step 1, in Algorithm 1) decides the amount of parallelism. Clearly, too many module bags (say, more than the number of processors) can slow down design-space exploration process. Too few module bags, on the other hand, mean under-
Algorithm 1 (Distributed Design-Space Search)

DFG: the data flow graph representing the behavior.
LIB: library of register-level modules available.

1. Generate the set $S$ of valid module bags.
2. for each valid module bag $B \in S$ do in parallel
3. begin
4. Schedule the DFG using the modules in the module bag $B$.
5. Estimate the area and speed of the design using the assigned schedule.
6. end
7. Choose the best schedule and the corresponding module set.

exploration of the design-space. If a module bag is known to subsume another module bag (not in set theoretic terms, but in terms of the quality of the schedules that these module bags lead to) then the second module bag should not be included in the collection $S$ of module bags. Some synthesis systems assume that there is only one module in the module library that can implement a given function [12, eg.]. Other systems, instead of attempting to find the best modules to use keeping the constraints in view, tend to use all the modules in the module library that can implement a given function [10, eg.]. The former approach is too restrictive and the latter approach could result in the usage of inefficient modules in the design where more efficient modules could have been used. In this paper, we present an efficient algorithm for generating a good set of valid module bags. This algorithm, discussed in detail in Section 3, admits only the most appropriate module bags into the set $S$ and disallows others; it is thus responsible for the amount of the design-space explored and the time taken by the tradeoff analysis phase.

We use a variation of force-directed list scheduling, due to Paulin and Knight, [12] as our basic scheduling algorithm. Our performance estimation procedures are extensions of (and are somewhat more rigorous than) those proposed by Jain, Milnar Kurdahi and Parker [4, 8]. Neither the basic scheduling algorithm nor the performance estimation procedures are discussed in detail in this paper. Interested readers may refer to [5] as well as the above references. Instead, the theme of this paper is the module bag generation algorithm and its effect on the quality of distributed design-space exploration. Included are some experimental results about the quality of the tradeoff analysis in comparison with other systems and the speedup results due to the use of a multiple processor system.

2 Inputs to the Exploration System

Our distributed design-space exploration system takes four inputs:

1. Data Flow Graph (DFG): The behavior specification is represented by a data flow graph (DFG) [15]. For the purposes of this paper we will not get into the details of this representation except to note that the nodes of the graph can represent either functions (such as $+$, $*$ etc) or control operations (such as case, call, etc). The function nodes which are eventually implemented by the functional units available in the library are of interest during the module bag generation phase. Figure 1 shows the DFG (data flow graph) representation of an Elliptic Wave Filter. This example was used as a benchmark at the High-Level Synthesis workshop in 1988. All the functions and edges (carriers) in this example have the same bit-width of 16.

   Given a DFG, we define the Set of Functions (SF) as the set of all function symbols used in the DFG. For the Elliptic Wave Filter example the set of Functions is $\{+, \cdot\}$.

   For each function in SF there should be at least one module that can implement the function in the module library; otherwise, the given DFG cannot be implemented using the given module library.

   For operations with different bit-widths are considered to be different functions. For example, 16-bit addition is viewed as a different function than 32-bit addition; these are denoted as $+_{16}$ and $+_{32}$ respectively. When there is no ambiguity, we drop the bit-width suffix.

2. Module Library: The module library contains (all the pertinent information about) the register-level modules that can be used in the design. Specifically, the module library contains several arithmetic-logic units each of which can implement one or more functions such as $+_{16}$, $\times_{16}$, $<_{16}$ etc. The set of functions supported by a library module is called the Intrinsic Function Set (IFS). Each
module also has two performance attributes: delay time and area. 2 Typical information available in the module library is shown in Table 1. Each function node in the data flow graph must be eventually mapped onto a module selected from the library such that the IFS (Intrinsic Function Set) of the module contains that function.

3. Area Constraint (A): Area constraint is a user-specified value. For the purposes of the paper it denotes the upper-limit on the arithmetic sum of areas of the modules in the module bag used to schedule the data flow graph. This constraint is used to delimit the modules selected from the module library. 4

4. Clock-Period Constraint (C): This user-specified constraint denotes the maximum delay of any module in the module bag used to schedule the DFG. Clock-period constraint is also used to delimit the modules selected from the library.

3 Generation of Valid Module Bags
A module bag is a collection of modules, with duplicates allowed, selected from the module library. A valid module bag (VMB) satisfies the following properties:

1. Property 1: For each function \( f \in SF \) (the set of functions in the given DFG), there exists a module \( M \) in the bag such that \( f \) is supported by \( M \), i.e., \( f \in IFS(M) \).
2. Property 2: The delay of each module in the bag is no more than the user-given constraint on the clock-period (C).
3. Property 3: The total area occupied by the modules in the bag should not exceed the area constraint (A) provided by the user. That is, \( \sum_{M \in VMB} M_i \leq A \).

A valid module set is a valid module bag which contains no duplicates.

The module bag generation algorithm has four major steps: (1) Identification of useful modules in the module library; (2) Generation of valid module sets (VMB's with no duplicates); (3) Estimation of the maximum number of copies of each useful module, required to implement the full parallelism available in the DFG; (4) Generation of valid module bags by enlarging the valid module sets generated in Step 2 by introducing duplicates up to the maximum numbers found in Step 3. These four steps are explained in detail in the following sections.

3.1 Identification of Useful Modules
Let the set of function types in the given DFG be SF. For each function type \( f \in SF \) the module library is consulted to determine the set of all the modules that support \( f \). From among these modules, those that violate the clock-period constraint (C) or the area-constraint (A) are rejected. Each remaining module is useful in the sense that it can be used to implement some function node in the DFG. An additional improvement is possible: if there are two useful modules which support exactly the same set of function types in the DFG, then the one with larger area can be ignored. The resulting module set is called the Useful Module Set (UMS). Algorithm 2 shows the procedure to compute the UMS, given a DFG (actually, the SF of the DFG is enough), a module library, and the area and clock-period constraints. In Algorithm 2, Supporting_Modules(f) is a library interface function which returns the set of all the modules in the library that implement \( f \).
For the Elliptic Wave Filter example, assume that the area constraint is 50 sq. mm. and the clock-period constraint is 100 ns. Using the module library information shown in Table 1, ALU, adder1 and adder2 can be used to implement the + operation; they all satisfy the clock-period constraint. However, since adder1 and adder2 are functionally equivalent, and adder1 has less area, adder2 is not included in the set of useful modules. Both the ALU and the multiplier are useful to implement the * operation.

3.2 Valid Module Sets

Recall that the valid module set is the same as a valid module bag except that it is a strictly a set (no duplicates allowed). A valid module set contains enough resources to implement all the functions in the DFG; it may not however be able to support the parallelism available in the DFG. Thus a design generated using a valid module set will be relatively slow, but will be small in terms of area. Such designs are usually called 'serial designs'.

Conceptually, the generation of valid module sets involves two steps in deriving the UMS (the useful module set computed in the previous step). From this powerset, all the sets that violate Property 1 or Property 3 are discarded to yield the set of all valid module sets. The following technique is used to reduce the time taken to enumerate the elements of the powerset of the UMS:

A critical module is a module in the UMS which supports a DFG function that is not supported by any other module in the UMS. Hence, all critical modules must be included in any valid module set. Based on this observation, the UMS can be partitioned into two subsets: the set of all critical modules and the set of all non-critical modules. The powerset of the set of non-critical modules is first generated. The set of critical modules is then added to each element of this powerset; each element is then tested for Property 1 and Property 3. Algorithm 3 shows the steps in generating the collection of all the valid module sets.

For the Elliptic Wave Filter example, the following valid module sets are generated: (ALU, multiplier), (adder1, multiplier) and (ALU, adder1, multiplier). All these sets satisfy the area constraint.

3.3 Potential Parallelism for Each Useful Module

A valid module set is sufficient to implement all the functions in the behavior. However, to exploit all the parallelism available in the data flow graph, multiple copies of some of the useful modules may be necessary. For a given module, just how many copies are necessary? The answer depends upon the amount of parallelism in the DFG, in terms of the maximum number of function nodes assignable to the same control step, where each function can be supported by a copy of that module.

Let SS denote the set of all possible valid schedules of the DFG. Given a schedule (S) (assignment of function nodes in the DFG to control-steps), a control step (CS), and a library module (M), let FS(M, CS) denote the bag of function nodes supported by M and assigned to CS. Clearly, for each $f \in FS(M, CS)$, $f \in IFS(M)$. Let $N_{S}(M, CS) = |FS(M, CS)|$ be the number of such function nodes. Maximum parallelism (MP), with respect to a module M, is then defined as the maximum such N. That is, $MP(M) = MAX_{S},CS\subseteq S | KS(N_{S}(M, CS))$, where KS is the maximum control-step number used in schedule S. The MP(M) (maximum parallelism wrt M) is an upper-limit on the number of copies of the module ever needed in any schedule of the DFG. How to determine MP(M) is the next question.

MP(M) can be rigorously determined by a clique partitioning formulation: First, a graph whose nodes are all the function nodes in the DFG covered by IFS(M) should be constructed. Edges should then be introduced such that any two nodes that have no direct or indirect data dependence are connected by an edge. Edges thus indicate the potential for assignment of the two nodes to the same control step. The number of nodes in the maximal clique of this graph is the value of MP(M).

Determining the maximal clique in a graph is an NP-complete problem. To avoid this, a good estimate of MP(M) can be obtained by considering the ASAP (As Soon As Possible) schedule of the DFG. Thus, we take MP(M) to be the maximum number of function nodes supported by IFS(M) and assigned to the same control step in the ASAP schedule.

Algorithm 4 shows the steps in determining the MP(M) values based on the ASAP schedule. MP values are determined for all useful modules in the set UMS using this algorithm.

On many occasions in practice, we discovered that the estimate of MP(M) obtained using Algorithm 4 works just as well as the rigorous approach based on clique partitioning; and, some times reduces the search-space in a nice

---

5Note that adder2 should not be deleted from the set of useful modules if a scheduling algorithm that permits chaining is used.

Algorithm 3 (Generation of Valid Module Sets)

```
SCM: Set of critical modules in UMS.
SNCM: Set of non-critical modules in UMS.
SVMS: Set of valid module sets.
SF: Set of function types in the DFG.

VALID-MODULE-SETS()
1. begin
2. SVMS ← {}  // initialize set of valid module sets
3. P ← powerset(SNCM)  // powerset of non-critical modules
4. for each set of non-critical modules $S \in P$ do
5. begin
6. $R ← (SCM \cup S)$  // union of critical and non-critical modules
7. $M \left( \forall f \in SF, \exists M \in R \text{ such that } f \notin IFS(M) \right)$
8. then SVMS ← SVMS ∪ R
9. end
10. return(SVMS)
11. end

CHECK-AREA-CONSTRAINT(MB)

MB: A bag of modules.

1. begin
2. area = 0
3. for each $M \in MB$ do
4. begin
5. area = area + area(M)
6. if (area > A) then return false
7. else return true
8. end
9. end

*ASAP scheduling is well-known and is not discussed here; see for example, [11].*
Algorithm 4 (Maximum Parallelism)

\[ M: \text{a library module} \]
\[ \text{IFS}(M): \text{intrinsic function set of } M \]
\[ \text{MAXIMUM-PARALLELISM}(M) \]
1. begin
2. max_count \leftarrow 0;
3. for \( i \leftarrow 1 \) to Max.ASAP.Cstep do
4. begin
5. count \leftarrow 0;
6. for each function node \( f \in \text{DFG} \) do
7. if \( (f \in \text{IFS}(M) \text{ and ASAP.Cstep}(f) = i) \)
8. then count \leftarrow count + 1;
9. max_count \leftarrow \max(count, max_count);
10. end
11. return max_count;
12. end

Algorithm 5 (Generation of Module Bags)

This is a recursive algorithm. The top level call on this algorithm must be of the form \text{MODULE-BAGS}(VMS) where \( VMS \) is a valid module set.

1. \text{MODULE-BAGS}(MS)
2. begin
3. \text{SMB} \leftarrow \text{};
4. for each \( M \in MS \) begin
5. \( Q \leftarrow \text{MODULE-BAGS}(MS \setminus \{M\}) \)
6. for \( i \leftarrow 1 \) to \( \text{MP}(M) \) do
7. \text{SMB} \leftarrow \text{SMB} \cup \{M, M, \ldots \text{i times}\}
8. else for each Bag \( \in Q \) do
9. \text{SMB} \leftarrow \text{SMB} \cup \{Bag \cup \{M, M, \ldots \text{i times}\}\}
10. end
11. return(SMB)
12. end

Table 2: Maximum Parallelism in EWF DFG

<table>
<thead>
<tr>
<th>Module ((M))</th>
<th>IFS((M))</th>
<th>MP((M))</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>+, *</td>
<td>4</td>
</tr>
<tr>
<td>adder, l</td>
<td>+</td>
<td>2</td>
</tr>
<tr>
<td>multiplier</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3: Some Sample valid module bags generated for the Elliptic Wave Filter Example.
4 Distribution of Module Bags and DFG

The valid module bags are then distributed across the processors. Each processor receives a copy of the DFG and one or more valid module bags. Each processor then executes the scheduling and performance estimation phases.

5 Scheduling

Our scheduling algorithm is an extended version of the force directed list scheduling (FDLS) developed by Paulin and Knight [12]. The extensions mostly pertain to the scheduling of control nodes in the data flow graph. Operations in the DFG are scheduled (assigned to control steps) under the resource constraints defined by the module bag.

6 Performance Estimation

For each schedule produced, the overall area and throughput time in terms of control steps are estimated. The area estimate consists of component (registers, wires, controller, ALUs) area estimates. The time estimate is calculated as the number of control steps to execute the longest path in the DFG. Some of our performance estimation techniques are similar to the ones proposed by Jain, Minar, Kurdi and Parker [4, 8, 6]; others are new techniques which yield more rigorous estimates [5].

7 Tradeoff Plots

Finally, the performance estimates produced in the processors are gathered and are presented to the synthesis system or to the user if so desired. The presentation is in the form of tradeoff plots. A variety of plots are produced to give the user a reasonable idea of the performance envelope.

For the Elliptic Wave Filter example, one of these plots is presented in Figure 3. The plot shows the variation in the overall area with respect to the control steps for each of the 29 valid module bags. The most desirable schedule (and module bag) may be chosen based upon the overall performance goals in terms of the number of control steps and/or the overall area.

Note that the design-space explored by the proposed algorithm subsumes that explored by the HAL synthesis system which also uses the force-directed scheduling algorithm as the basis. (See [12] for the design-space HAL produces for the same Elliptic Wave Filter example.) This, of course, is to be expected due to the rather extensive set of resource bags generated and considered by our algorithms.

8 Implementation and Speedup

The proposed algorithms for distributed design-space exploration have been implemented as part of the DSS (Distributed Synthesis System) [14]. DSS executes on the ESK (Experimental Systems Kit) multiple processor machine developed at the MCC [3]. The version we used has 16 nodes connected in mesh-configuration. There is no shared memory; all communication is through high-speed links. The speed of the message fabric is such that from the programmer's point of view, the mesh configuration is unimportant; the message communication time between any two processors (regardless of whether the two processors are neighbors) is about the same. DSS is implemented in the extended C++ supported by the ESK system. During the design-space exploration phase, load balancing is attempted by distributing equal number of bags to each processor.

Three examples were considered for experimentation. These examples are described in detail in [17]. Briefly, the first example is the Move Machine (70 lines of VHDL code). Move machine is a control dominated machine. The Move Machine has instructions to move data between memory and registers. The second example is the Elliptic Wave Filter (50 lines in VHDL) which is an arithmetic-dominated machine. The last example is the Viper microprocessor [17] which is a substantially large example (400 lines of VHDL code). Viper is an instruction set processor containing both control and arithmetic statements.

Figure 4 shows the variation in execution time (from parsing the specification to producing the performance plots) with respect to the number of processors. For the Viper and Elliptic Wave Filter examples, close-to-linear speedup is observed up to about 8 processors, after which

---

Algorithm 6 (Generation of Valid Module Bags)

```
VALID-MODULE-BAGS(svms)
1. begin
2. svmb = null
3. for each vms e svms do
4. for each mb e MODULE-BAGS(vms) do
5. if CHECK-AREA-CONSTRAINT(MB)
6. then svmb = svmb u MB
7. return svmb
8. end

Figure 3: Performance Plot (Ctrl-steps vs Area) for Elliptic Wave Filter

---
speed-up tapers off. This occurs due to the communication overheads which start dominating the overall time. For Move Machine description only 3 valid module bags were generated, which explains why there was no speedup.

9 Discussion

The Elliptic Wave Filter example represents a typical arithmetic-dominated machine and the Move Machine and Viper represent control-dominated machines. The full set of performance trade-off plots for the Elliptic Wave Filter, Move Machine, and Viper example can be found in [5].

The DFG representation is not as simple as presented in this paper; it is quite elaborate to accommodate various idiosyncrasies of VHDL. These idiosyncrasies have many ramifications on the scheduling algorithms, particularly in force calculations and while making deferral decisions. We have made non-trivial extensions (eg. how to schedule wait nodes, how to schedule parallel loop blocks, how to schedule concurrent procedure invocations etc.) to the basic force-directed scheduling algorithm to account for all the features of the DFG. This scheduling algorithm is discussed in detail in [5].

As a part of the ADAM work at USC, Rajiv Jain [4] developed lower-bound area-delay curves which give some indication of the nature of useful module bags. The basic idea (that of generating many module bags) we used is similar but the techniques however are different. The amount of design-space explored in our case is some what more extensive and covers many interesting trade-off points. Also, the performance estimation procedures we have developed (not presented in this paper) are aimed at estimating the actual values instead of determining the lower bounds.

All the algorithms presented in this paper make extensive use of set operations (union, intersection, power-set etc). In our implementation sets are implemented as C++ classes (resulting in a set data type). Within the class, the methods (implementing set operations) are implemented using doubly linked lists and are very efficient in execution time. In our algorithms, we have suggested that all module sets/bags are first generated and then filtered out based on the area constraints. In the implementation, however, the generation and filtering of module sets/bags happens hand-in-hand to improve the space-efficiency of the algorithms. In this paper, for clarity, we have not presented the algorithms at that level of detail. The exact data structures and algorithms used along with their complexity analyses can be found in [5].

Careful analysis of the algorithms presented in this paper will show that there is considerably more parallelism that can be exploited (for example, most of the for-all and for-each loops can be executed in parallel after slight modifications). It remains open to investigate whether it is worth, in terms of communication cost vs. potential speedup, attempting to exploit such fine-grain parallelism.

References


