EXPLORA - GENERIC DESIGN SPACE EXPLORATION DURING EMBEDDED SYSTEM SYNTHESIS

Frank Cieslok, Heinrich Esau, Jürgen Teich
Computer Engineering Lab (DATE), University of Paderborn
Warburger Straße 100, 33098 Paderborn, Germany
email: teich@date.upb.de

The need for design space exploration on different levels of abstraction during synthesis of electronic systems has received wide attention recently. Unfortunately, there are almost no tools available on the EDA market that allow a designer to enhance his synthesis tool suite by design space exploration capabilities easily. A versatile tool for design space exploration must be targetable to different synthesis tools. Also, different optimization (exploration) algorithms should be able to be connected to such a versatile tool. Here, we present an approach that enables design space exploration with support to couple different exploration algorithms and synthesis tools. Our JAVA based tool called EXPLORA is also able to visualize the exploration results and can be adapted to new problems and abstraction levels within hours.

1. Introduction

The goal of this work is to design a tool that enables the design space exploration during the design of embedded systems, i.e., during hardware/software codesign. Only recently has this problem been considered as a need in order to estimate the quality of different design alternatives in early design phases and propose the designer a number of optimal solutions with different characteristics (e.g., speed versus cost, power, etc.) from which he/she may decide to choose and refine one solution to the final product. In this area, multiobjective optimization problems have to be solved. An approach that uses evolutionary algorithms for design exploration of Pareto-optimal fronts can be found in [1] for system-level synthesis (a generalization of hardware/software partitioning), or for optimal code synthesis for DSP processors from data flow graphs [4], [6], [5].

Unfortunately, existing tools are either too specialized to be used for different synthesis problems, too tightly coupled to tools that are used for evaluating the quality of a design point, heavily dependent on architecture assumptions, design abstraction, etc. such that a reuse of such tools is simply impossible. Here, we

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35409-5_23 present the structure of a versatile tool for design space exploration called EXPLORA that may be easily incorporated into any level of design abstraction. The flexibility results by addressing the following problems and requirements:

- Formal (functional) quantification of the nature of design space exploration processes involving synthesis tasks.
- Clear separation between
    - Problem-specific parameters (e.g., dimension of exploration space, metrics, cost function, etc.)
    - Independence of synthesis algorithm and implementation language
    - Independence of optimization (exploration) algorithm and implementation language
    - Visualization support
- Finally, it should be easy to couple such a design space exploration tool to existing environments.

First, we give a characterization of design space exploration processes during embedded system synthesis. In Section 3, we present the mathematical background to formalize the process of generic design space exploration. In Section 4, the structure of EXPLORA is introduced. Finally, in Section 5, we present a typical scenario for coupling the well-known Synopsys behavioral compiler, see, e.g.,[3], for design space exploration during high-level synthesis as a case study.

2. Characterization of Design Space Exploration Processes

2.1 Structure of design space exploration processes

Two of the basic requirements of a flexible tool for design space exploration are a) the exchangeability of the optimization algorithm that is used for exploration of the design space and b) its adaptability to different synthesis tools that may be used to compute the quality of points in the design space concerning cost, speed, and other metrics. A natural distinction is to split the process of design space exploration into three main tasks: The first one contains all synthesis tool specific behavior, the second concerns the optimization algorithm for evaluating solutions and selecting new design points in the design space with the goal to obtain a high diversity (covering) of optimal points. The third module manages the exploration process itself using handles to the other modules.

Since the optimization algorithm needs a cost function which rates a given result produced by the synthesis tool, it is useful to split the module with the optimization algorithm into a second module for computing the cost function (see Fig. 1). This way, the cost function can easily be changed by the user, too, without the need to exchange the optimization algorithm.

Figure 1 shows the structure of a tool for generic design space exploration. The exploration manager starts a synthesis tool with certain parameters and obtains the synthesis results of this tool. This result is forwarded then to the optimization algorithm that is used during the exploration. This algorithm provides the next parameter set(s) (new design point(s)) to explore. The optimization module in turn uses a cost function which rates a given result. Finally, it is also desirable to have a graphical user interface (GUI) that gathers and visualizes the progress and results during the exploration process.

This section deals with the description of the modules in Fig.1. We will describe their behavior using functional abstractions. First, some explanations are in order.

3.1 Explanations

Before describing the behavior of the exploration program modules, it must be specified which kind of data is exchanged. Figure 1 shows the three different data structures parameters, result and costs. The parameters-object characterizes a design point and can be described by a set of parameters needed by the synthesis tool to perform a synthesis.

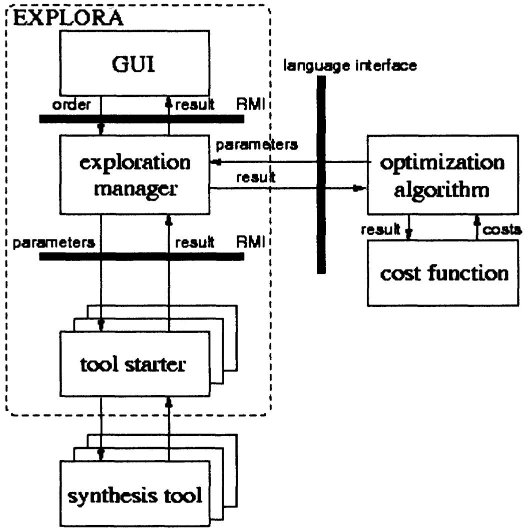


Figure 1: Structure of a program for design space exploration

A result-object represents that part of the output of a synthesis tool which is important to control the exploration flow and which is needed by the user to rate this result. Basically, it consists of a set of quantities representing the properties of the synthesis tool output.

Example 3.1

Consider the abstraction-level of high-level architectural synthesis, for instance, and the Synopsys Behavioral Compiler [3]. Among others, this tool has the option parameter io_mode with the possible values cycle_fixed, free_float and superstate.[1] It further needs a file with the behavioral description of a certain design, e.g., a VHDL or Verilog file. The filename is interpreted as a parameter, too, so the

[1] The values of this parameter denote whether I/O-operations of a given design must be scheduled in certain fixed cycles, in arbitrary cycles or respecting a certain partial order, respectively.

Behavioral Compiler should produce a synthesized data path plus controller when invoking it with these two parameters and proper values. In order to evaluate the quality of such a synthesized design, the Behavioral Compiler or any synthesis tool typically outputs some result and log files, and special postprocessing steps are usually needed in order to extract the important quality metrics needed for design evaluation such as the area requirement of the synthesized design, its required latency, minimum possible clock cycle time, etc. These are typical synthesis results needed for rating the quality of a design during design space exploration. Each parameter set producing a different design is called a design point.

The costs-object may often be simply described by a tuple of real numbers which must be properly interpreted by the optimization algorithm to rate a certain result. With these explanations, the task of each exploration program module can be defined using functional abstractions of each module.

3.2 Tool starter module abstraction

Each synthesis tool needs some input files to perform a synthesis and produces some output files as a result. During the exploration, possibly not the whole input should be modified (e.g., the design specification stays the same) and not all parts of the output (e.g., command log files) should be taken into consideration. So the task of the tool starter in Fig. 1 is to produce the complete input needed by the synthesis tool given a parameter set and to extract the desired result quantities from the output returned by the synthesis tool after its completion. The tool starter sits on top of the corresponding synthesis tool and behaves like an independent tool to its invoker, so its behavior can be described by a function as follows:

Let a certain synthesis tool starter have n parameters $p_1,...,p_n$ with the domain P_i for the parameter $p_i, i = 1,...,n$. Hence, a design point may also be characterized by an n-tuple $\vec{p}$ without loss of generality. Let the corresponding tool produce m result quantities $q_1,...,q_m$ that may be represented by a tuple $\vec{q}$. Let Q_i be the domain of the quantity $q_i, i = 1,...,m$. If there are no other constraints, then $P = P_1 \times P_2 \times ... \times P_n$ is called the *design space* $Q = Q_1 \times Q_2 \times ... \times Q_m$ the result space. The behavior of this synthesis tool starter can be described by a function

$$synth: P \rightarrow Q \qquad (1)$$

Example 3.2

Let io_mode and vhdl_file in the previous example be the parameters of the Synopsys Behavioral Compiler, the tool starter would expand this parameter set and extract proper values for the area and latency time of a synthesized design afterwards. Suppose the name of the VHDL file is design.vhdl, the domain of the function synth would be $P = \{design.vhdl\} \times \{cycle\_fixed, superstate, free\_float\}$

and the range of result values $Q = \mathbf{R}^+ \times \mathbf{R}^+$. For each valid VHDL file and for each value of io_mode the function synth would produce a pair of values for the area and the latency time.

3.3 Cost function abstraction

For rating a synthesis result, the optimization algorithm needs a cost function which builds a tuple of costs from a tuple of result quantities. Let $c_1,...,c_l$ be l cost quantities. Let C_i be the value range of cost parameter $c_i, i = 1,...,l$. The range of values of the resulting cost tuple $\vec{c}$ is $C, C = C_1 \times C_2 \times ... \times C_l$. The cost function is then

$$\text{cost: } Q \rightarrow C$$

(2)

Example 3.3

Continuing the previous example, one cost function would be the weighted one dimensional function $(l = 1, m = 2)$

$$cost(area(\vec{p}), latency(\vec{p})) = 0.7 * area(\vec{p}) + 0.3 * latency(\vec{p}).$$

This way, the cost function would weight the area of a design point $\vec{p}$ more than its latency time and thereby force the exploration in a direction which rather would produce smaller than faster designs.

Example 3.4

Let us consider, without loss of generality, a multi-objective minimization problem with m result parameters for each design point $\vec{p}$ of dimension n and l objectives. Let $\vec{q} = synth(\vec{p})$. Minimize $\vec{c} = cost(\vec{q}) = (cost(\vec{q}),...,cost(\vec{q}))$

(3)

where $\vec{q} = (q_1,...,q_m) \in Q \; and \; \vec{c} = (c_1,...,c_l) \in C$ are tuples with $c_i = cost_i(\vec{q}), i = 1,...,l.$ $\vec{a} \in Q$ is said to dominate $\vec{b} \in Q$ (also written as $\vec{a} \succ \vec{b}$) iff

$$\forall i \in \{1,...,l\}: cost_i(\vec{a}) \leq cost_i(\vec{b}) \quad \wedge$$

(4) $$\exists j \in \{1,...,l\}: cost_j(\vec{a}) < cost_j(\vec{b})$$

$\vec{a}$ covers $\vec{b}$ $(\vec{a} \succeq \vec{b})$ iff $\vec{a} \succ \vec{b}$ or $cost(\vec{a}) = cost(\vec{b})$. All design points $\vec{p}_i \in P$ with the property that $q_1 = synth(\vec{p}_i)$ is not dominated by any other $q_j = synth(\vec{p}_j)$, $\vec{p}_j \in P$, are called *nondominated.* Pareto-optimal points are the nondominated design points of the entire search space P.

For design space exploration, a useful cost function is to let $cost(\vec{x})$ be equal to the number of design points explored so far that dominate $\vec{x}$. Hence, after the exploration, all explored points with cost zero are (approximations of) Pareto-optimal points.

3.4 Optimization module abstraction

The task of the optimization module is to produce a set of new parameter sets (design points) to explore next given a set of design points and their ratings.
If a given optimization algorithm needs υ different synthesis results to generate ω new design points, then the behavior of the optimization module could be described by a function $opt : (PxC)^{\upsilon} \rightarrow P^{\omega}$
(5)

Example 3.5
Consider an optimization (exploration) algorithm that is population-based, e.g., a variant of an Evolutionary Algorithm that simply selects the best result from n given result objects using a certain cost function and produces n identical copies of this optimal design point as offspring, however, with random variations in its parameters. In the next iteration of the exploration, this set of mutated design points would be used by the tool starter and produce new synthesis results. This way, the algorithm would implement the function $opt : (PxC)^{n} \rightarrow P^{n}$ with $C = cost(synth(P))$.

Example 3.6
A well-known local search technique for solving hard combinatorial problems is simulated annealing [2]. Here, the algorithm decides based on a single result object which new design point in the neighborhood will be investigated next: $opt : (PxC)^{1} \rightarrow P^{1}$. Hence, the exploration describes simply a path in the design space.

3.5 Exploration manager

The exploration manager has just administrative tasks. To perform the exploration for a given synthesis tool, the exploration manager has to properly invoke the corresponding functions. As exploration is an iterative process, the obvious idea is to have a loop somewhere in the manager module that rates previously generated results using the cost function $(cost)$, starts the optimization module with these ratings and the corresponding parameter sets (opt), and calls $synth$ for each new design point to be explored. Conceptually, this means the successive execution of the functions $cost$, opt and $synth$ in each iteration. Generally, this isn't straightforward, since in the case if opt needs υ different (cost, parameter) tuples to produce ω new parameter sets, the function $cost$ must be invoked υ times, and after that $synth$ must be executed ω times. Additionally, the values υ and ω are generally not constant, and the process of the exploration must be take care of that. In the following, we describe how this problem is solved in the implementation of EXPLORA.

4. EXPLORA - a Tool for Versatile Design Space Exploration

This section deals with the implementation of EXPLORA. The main issues are the principal process of exploration and the structure of the exploration manager module.

4.1 EXPLORA program structure

In addition to the previously described exploration manager, EXPLORA consists of a GUI (graphical user interface) where the user can visualize the exploration results and choose a single implementation for further evaluation and also a JAVA based remote method invocation (RMI) interface between the GUI and the exploration manager providing the possibility for different users to interact with the exploration manager at the same time. That way a single instance of an exploration manager can care for the administration of all accessible tool starter instances on a network. This central instance is responsible to apportion all incoming exploration tasks to the registered tool manager services.

To be able to execute several instances of a synthesis tool simultaneously on different computers of a network in order to speed up the exploration process, there is a second RMI interface between the exploration manager and the different tool starter instances. Although belonging to the EXPLORA tool suite, the tool manager objects are started on the machines providing the tool services needed for exploration. All tool starter instances are registered at the central exploration manager service.

To provide a way for flexible optimization algorithm adaptation, a language interface is introduced as shown in Fig. 1 providing the possibility to implement these algorithms in arbitrary programming languages. This way it would, for instance, be possible to generate the cost function implementing object in a script language like Python which would be reconfigurable on-the-fly by the designer without the need of recompiling the EXPLORA tool suite.

4.2 EXPLORA management process

The exploration manager is able to serve several GUIs simultaneously by using one optimization module for each GUI and distributing the generated exploration tasks to several tool starters. Task queues are used to be able to serve requests coming from different users without resource collisions. The management cycle consists of a periodic thread which receives a new parameter set from the set generating optimization module if available and adds it to the appropriate task queue. The parameter set is forwarded to a tool as soon as the related tool instance is able to process a new request. Afterwards, the result is forwarded to the optimization module for generation of the next parameter set and at the same time to the GUI to be displayed.

4.3 Graphical User Interface

One of the main requirements for the exploration program is the ability to visualize the exploration results. Therefore, the graphical user interface of EXPLORA shown in Fig. 2 mainly consists of a two dimensional sheet where for each explored point $\vec{p}$, its result tuple $\vec{q} = synth(\vec{p})$ is displayed. The sheet is auto-scaling depending on the explored result values discovered so far. There is an additional text field (left) where the parameters of a currently selected result object can be displayed in textual form. For result tuples of dimension $l \geq 3$, the user must choose 2 out of l quantities to be displayed.

5. Case Study: Design Space Exploration During High-Level Synthesis

Note that the following example of high-level synthesis is just one example where the above concepts have been applied successfully in the context of embedded system synthesis. Others are hardware/software partitioning (system-level exploration) [1] and the automatic exploration of task mappings in the context of massively parallel processor arrays.

5.1 Problem specification

The workflow of the exploration process using EXPLORA will be demonstrated by the synthesis of a single-chip embedded system design based on behavioral VHDL with an FPGA as target architecture. The typical design flow includes three steps:

1. Specification and implementation of the design in VHDL.
2. Compilation into a netlist format via an RT-level synthesis tool.
3. Technology mapping, place and route using another tool provided by the FPGA vendor.

In this case study, the *Synopsys Behavioral Compiler* is used for compiling the VHDL code into a netlist format. The FPGA data stream is created by the *Xilinx Alliance Tools* for a *Xilinx 4028EX* FPGA as target architecture.

Both tool suites provide a wide range of possible options strongly influencing the design quality. The designer has to decide, e.g., what scheduling strategy the VHDL compiler should use for minimum latency with unknown effects on the chip area or if the mapping tool should care most for speed, area, or look for a balanced solution. Additionally, it must be possible to explore in which way certain design constraints like net delay constraints specified in so-called user constraint files influence the quality of the results. Supposing that a complete design run from code compilation to a downloadable data stream may take some minutes or more, testing all possible parameter sets by hand is not feasible. Instead, EXPLORA will be used to explore solutions in a stand-alone program run over night using all workstations available to it on the net and then let the designer choose the most convenient one afterwards.

Example 5.1

As an example, we consider a well-known differential equation benchmark from high-level synthesis. The behavioral VHDL specification to solve the differential equation $y''+3xy'+3y = 0$ in the interval $[x, a]$ using step size dx and initial values $y(0) = y, y'(0) = u$ using the Euler method is given as follows:

```
ENTITY dgl IS
  PORT(x_in,y_in,u_in,dx_in,a_in: IN REAL;
       activate: IN BIT; y_out: OUT REAL);
END dgl;
ARCHITECTURE behavioral OF dgl IS BEGIN
  PROCESS (activate)
    VARIABLE x, y, u, dx, a, x1, u1, y1: REAL;
  BEGIN
   x := x_in; y := y_in; u := u_in; dx := dx_in; a := a_in;
   LOOP
    x1 := x + dx;
    u1 := u - (3 * x * u * dx) - (3 * y * dx);
    y1 := y + (u * dx);
    x := x1; u := u1; y:= y1;
    EXIT WHEN x1 > a;
   END LOOP;
   y_out <= y;
 END PROCESS;
END behavioral;
```

After functional test of the VHDL specification, EXPLORA can be started. It generates a set of design parameters and inserts them into a *makefile* which provides a set of rules for starting the appropriate tools with correct syntax and order of execution. Additionally, this makefile generates the constraint files needed by the synthesis tools. After a single synthesis run has been completed, the same makefile cares for result extraction out of the log files generated during synthesis using standard UNIX shell tools and hands these results back to EXPLORA which then adapts the parameter set for the next synthesis run if necessary. Multiple synthesis processes can be executed in parallel on different machines if available.

5.2 Optimization algorithm

In this case study, the following five parameters of the synthesis process are varied during exploration:

Parameter	Affected tool	Possible values
I/O mode	Synopsys BC	superstate, free_float
Area opt.	Synopsys BC	yes,no
Cover mode	Xilinx Mapper	area,speed,balance,none
Logic opt. effort	Xilinx Mapper	normal,high,none
Delay-based cleanup passes	Xilinx Placer	[0,5]

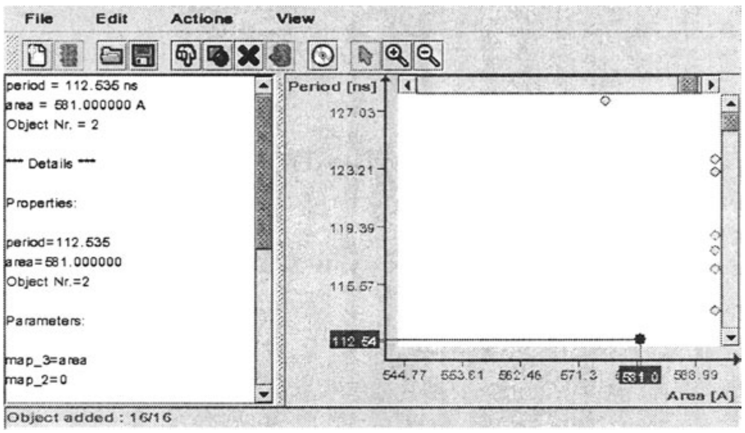
Here, a simple *greedy algorithm* is used to sequentially generate parameter sets out of the possible 288 parameter combinations as follows:

```
let $mincost := \infty$
Generate random initial parameter set $\vec{p} = (p_1,\dots,p_n)$
loop
  let $changed := false$
  for $i = 1 \dots n$ do
  Randomly change parameter $p_i$
  let $\vec{q} := synth(\vec{p})$
  if $cost(\vec{q}) < mincost$ then
    Keep last parameter set change
    let $mincost := cost(\vec{q})$
    let $changed := true$
    else
      Reject last parameter set change
    end if
  end for
  if changed := false then
    Exit loop
  end if
end loop
```

More sophisticated optimization algorithms like simulated annealing or evolutionary algorithms can easily be incorporated.

5.3 Results

Fig. 2 shows a screen shot of EXPLORA after 16 synthesis runs. The parameters important for rating the synthesis results are estimated area consumption (displayed on the x axis) and minimum possible FPGA clock period (displayed on the y axis). Additionally, the latency of the design could be extracted from the output of the scheduler and displayed alternatively together with any other result dimension.



```
period = 112.535 ns
area = 581.000000 A
Object Nr. = 2

---- Details ----

Properties:

period=112.535
area=581.000000
Object Nr.=2

Parameters:

map_3=area
map_2=0
```

```
Object added : 16/16
```

Figure 2: Result of the exploration process displayed in EXPLORA

In Fig. 2, two Pareto-optimal points are shown:

No.	Max. FPGA clock freq.	Area units
1	8.88 MHz	581
2	7.83 MHz	576

The parameter set used to generate solution (1) was $P=$ (superstate, yes, balance, high, 2). The remaining 14 results require a larger chip area of 592 units. 6 solutions are not visible in the display shown in Fig. 2 because they are covered by other solutions having the same result values.

One synthesis run takes approximately 8 minutes on a Sun UltraSparc 60. So the sequential exploration process can be done in about two hours and leads to much better results than randomly choosing tool parameters. An exhaustive search of about

39 hours may also be feasible. If the optimization algorithm supports parallel evaluation of different solutions like, e.g., evolutionary algorithms, the exploration process can be accelerated using several workstations.

6. Conclusions

We have presented an approach for flexible exploration of design spaces that are spanned by parameter ranges of synthesis tools involved during the development of embedded systems. Due to space limitations, we were only able to give one example of a typical abstraction level, namely high-level architectural synthesis, where the concepts of EXPLORA apply. Other levels include the system-level where hardware-/software implementation decisions are taken and tasks mapped to either hardware of software [1] or the real-time software synthesis level where a set of tasks has to be scheduled under real-time constraints. The main strength of our approach is the flexibility in exchanging synthesis tool, cost function and optimization algorithm at each level.

In the future, we would like to show that using the presented approach, also hierarchical design space exploration becomes possible. For example, a tree of differently configured EXPLORA processes may be started at different levels of abstraction and synchronized appropriately such that exploration results on lower levels of abstraction may be passed to parent processes, e.g., by the introduction of appropriate combining cost functions (sum, maximum, etc.).

References

[1] T. Blickle, J. Teich, and L. Thiele. System-level synthesis using Evolutionary algorithms. J. Design Automation for Embedded Systems, 3(1):23-58, Jan. 1998.

[2] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. Science, 220(4598):671-680, 1983

[3] D. W. Knapp. Digital System Design Using the Synopsys Behavioral Compiler. Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[4] J. Teich, E. Zitzler, and S. S. Bhattacharyya. 3D exploration of software schedules for DSP algorithms. In Proc. CODES'99, the 7th Int. Workshop on Hardware/Software Co-Design, pages 168-172, Rome, Italy, May 1999.

[5] E. Zitzler, J. Teich, and S. Bhattacharyya. Evolutionary Algorithms for the Synthesis of Embedded Software. J. IEEE Trans. on VLSI Systems, Vol. 8, No.4, pp. 452-456, August 2000.

[6] E. Zitzler, J. Teich, and S. S. Bhattacharyya. Evolutionary Algorithm Based Exploration of Software Schedules for Digital Signal Processors. Proc. GECCO'99, the Genetic and Evolutionary Computation Conference, Orlando, U.S.A., July 1999.