

# **CS812: High-level Design & Modelling of Digital Systems**

## **A report on Hardware/Software Partitioning**

Submitted by  
Amit Kumar Gupta (2001VLS007)  
Ram Babu Roy (2001VLS022)

Instructors  
Prof. Anshul Kumar  
Prof. M.Balakrishnan



**Department of Computer Science and Engineering  
Indian Institute of Technology, Delhi**

## Abstract

Complex systems can no longer be effectively designed without considering the interaction of the software and hardware domains early in the design process. It involves the development of an environment for the co-design of concurrent heterogeneous, hardware and software systems enabling the complete exploration of the physical design space between the two domains. This paper presents some heuristics for automatic hardware/software partitioning of system level specifications. Partitioning is performed at the granularity of blocks, loops, subprograms, and processes with the objective of performance optimization with a limited hardware and software cost. We define the metric values for partitioning and develop a cost function that guides partitioning towards the desired objective. We consider minimization of communication cost and improvement of the overall parallelism as essential criteria during partitioning. There are various heuristics for hardware/software partitioning. We give a brief idea of the simulated annealing algorithm, tabu (forbidden) search algorithm, energy-conscious HW/SW-partitioning and a detailed discussion on the hierarchical evolutionary approach. In hierarchical evolutionary approach, we apply a hierarchical structure and dynamically determine the granularity of tasks and hardware modules to adaptively optimize the solution while keeping the search space as small as possible. *Imprecisely Specified Multiple Attribute Utility Theory* has the advantage of constraining the solution space based on the designer's preference, but suffers from high computation overhead. We also discuss the technique to reduce this overhead.

## Introduction

New tools which extend design automation to system level have to support the integrated design of both the hardware and software components of an embedded system. The input specification accepted by such design tools describes the functionality of the system together with some design constraints and is typically given as a set of interacting processes. Many embedded systems have strong requirements concerning the expected performance. Satisfaction of these performance constraints can frequently be achieved only by hardware implementation of some components of the specified system. This can be realized by using dedicated ASICs (application specific integrated circuits) or FPGAs (field-programmable gate arrays) as part of the implementation architecture. Another solution in order to meet design requirements is to develop an ASIP (application specific instruction-set processor) with a carefully selected instruction-set and architecture, which produces the required performance for the given application.

Hardware/software partitioning, is an important step in hardware/software co-design. HW/SW partitioning actually means the selection of appropriate part of the system for HW/SW implementation i.e. which system tasks should be realized in which hardware modules. This has crucial impact on cost and overall performance of the system. For small systems partitioning can be done by designer's experience and intuition. For large systems it needs high performance heuristics and CAD tools. It is clearly critical to board-level designs and is becoming increasingly important in system-on-a-chip (SOC) designs as more and more intellectual property (IP) components are available. The objective of hardware/software partitioning is to search for an assignment of system tasks to hardware modules which not only satisfies the constraint (such as timing), but also optimizes desired

quality metrics, such as cost, power, and so on. This type of constrained optimization problem has been shown to be NP-hard.

Various steps followed in HW/SW partitioning are: problem formulation, performance optimization, cost minimization, meeting all design constraints. Some performance constraints are met only in HW implementation, so for this dedicated ASIC/FPGA is used. Partitioning approaches differ in Initial specification of the level of granularity, degree of automation, cost function and the partitioning algorithm used. Therefore the main partitioning issues are: the specification of abstraction level, granularity, system-component allocation, metrics and its estimations, objective and closeness functions, partitioning algorithms, output, flow of control and designer's interaction. The metrics generally used are: monetary cost, execution time, communication bit rates, power consumption, area, pins testability, reliability, program size, data size, memory size etc.

## The Co-Synthesis Environment

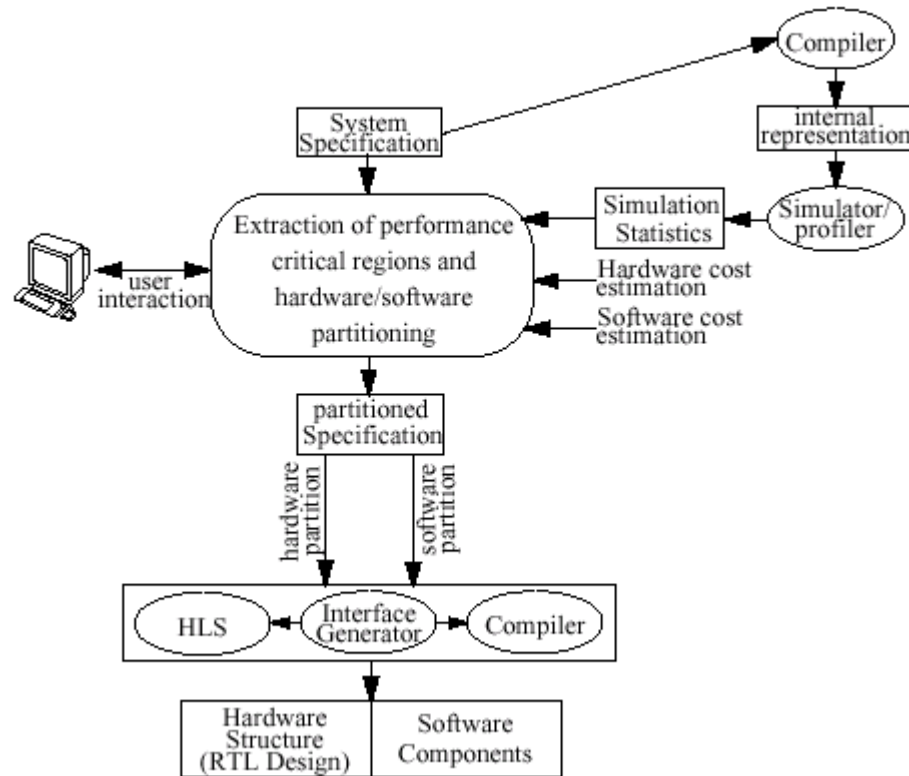


Fig. 1. Overview of the co-synthesis environment

An overview of the structure of the hardware/software co-synthesis environment is depicted in Figure 1. The input specification describes system functionality without prescribing the hardware/software boundary or implementation details. This specification is formulated as a set of processes interacting via messages transmitted through communication channels. This specification is further decomposed into units of smaller granularity. The partitioning algorithm generates as output a model consisting of two sets of

interacting processes. The processes in one set are marked as candidates for hardware implementation, while the processes in the other set are marked as software implementation candidates. The main goal of partitioning is to maximize performance in terms of execution speed.

## The partitioning steps

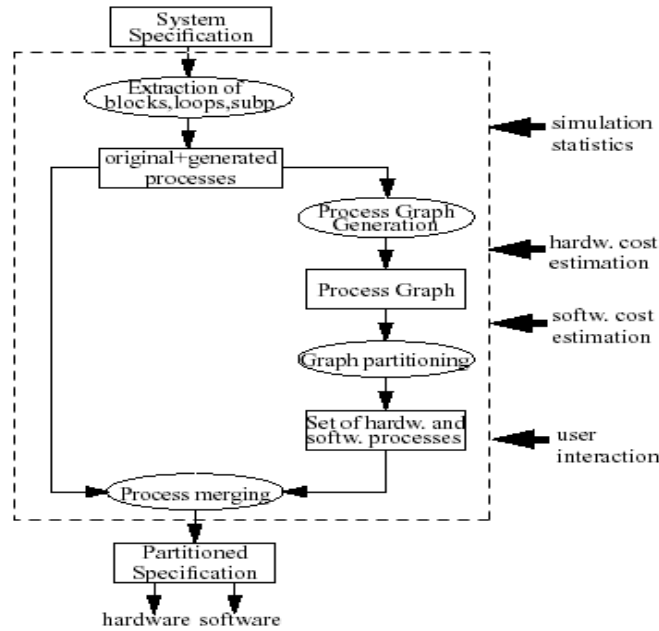


Fig. 2. The partitioning steps

1. *Extraction of blocks of statements, loops, and subprograms*: processes that are responsible for most of the execution time spent inside a process (regions with a large computation load). Candidate regions are typically loops and subprograms, but can also be blocks of statements with a high computation load. The designer guides identification and extraction of the regions and decides implicitly on the granularity of further partitioning by identifying a certain region to be extracted (regardless of its computation load) assigning hardware or software partition and by imposing boundary values.

2. *Process graph generation*: During the second step an internal structure, called the process graph, is generated.

3. *Partitioning of the process graph*: We formulate hardware/software partitioning as a graph partitioning problem performed on the process graph.

4. *Process merging*: During the first step one or several *child* processes are possibly extracted from a *parent* process. If, as result of step 3, some of the child processes are assigned to the same partition with their parent process, they are, optionally, merged back together.

## Partitioning algorithms

Partitioning is mainly divided into two classes: structural partitioning and functional partitioning. Structural partitioning can be easily mapped to a graph partitioning problem. In this, the size/performance tradeoffs are difficult due to large number of objects. Therefore, it is limited to only hardware designs. Functional partitioning divides the functionality into non divisible pieces called functional objects. The advantages of this are: easier size/performance tradeoffs, smaller no of objects, possibility of both the hardware and software solutions.

Basic partitioning algorithms are: random mapping, hierarchical clustering, multistage clustering, group migration, ratio cut, Kernighan-lin algorithm, simulated annealing, genetic evolution, integer linear programming. Hardware software partitioning algorithms generally used are: greedy algorithms, hill-climbing algorithms, binary constraint search (BCS) HW/SW partitioning, energy-conscious HW/SW partitioning, preference-driven hierarchical HW/SW partitioning, simulated annealing, tabu search etc.

In hardware/software partitioning both tasks and hardware modules can have different granularities. Coarse granularity means the tasks or hardware modules contain large amounts of behavioral or functional specifications while finer-granularity means tasks or hardware modules contain smaller amounts. A number of papers have dealt with the partitioning problem. They differ by the levels of abstractions, the target architecture, and/or the search algorithms used, but these approaches all presume a fixed granularity of tasks and hardware modules. This presumption might be sufficient whenever the behavior of a system and the functionality of the hardware objects are relatively simple, but for a large-scale, real-time embedded system, far more detail is typically required. These specifications are not easy to be satisfied when partitioning is performed at a coarse grain level. Conversely, with finer-grain partitioning, the cost to solve the partitioning problem increases exponentially with the number of tasks and hardware modules. Therefore it is believed that a mixed granularity representation provides a reasonable compromise.

Generally, hierarchical approaches are well-suited for complicated problems because complex systems can be hierarchically decomposed into a set of simpler systems, which is easier to deal with. Researchers have adopted hierarchy for hardware/software partitioning problems in a variety of ways. However, the hierarchal structure in the embedded system can be exploited more efficiently and aggressively. In this paper, we give a brief idea of the simulated annealing algorithm, tabu (forbidden) search algorithm, energy-conscious HW/SW-partitioning and a detailed discussion on the hierarchical evolutionary approach. The hierarchical evolutionary approach incorporates hierarchical structures for both tasks and hardware modules. We employ an evolutionary algorithm (EA) to efficiently handle the hierarchical tasks and hardware modules. In the hierarchical approach, the search space is maintained as small as possible. Partitioning always starts from a coarse level and switches to a finer level only when it becomes difficult to find a satisfactory solution. The key to our partitioning algorithm is to search for the optimized solution by partitioning objects with dynamically determined granularity.

Another challenge in solving the hardware/software partition problem is how to effectively and efficiently rank solutions which often have conflicting design criteria. For example, minimizing power consumption frequently requires a reduction in clock speed. The weighted-sum approach is intuitive and easily implemented, but the selection of precise weights is not always straightforward. Another method, the Pareto optimal ranking, is based on the assumption that all Pareto optimal solutions are equally preferable—a situation that does not hold in many real world applications. The *Imprecisely Specified Multiple Attribute Utility Theory* (ISMAUT) combines the advantages of both a weighted-sum method and a Pareto optimal ranking algorithm. With ISMAUT, comparison of two alternative designs can be directed by the preference of the designer. However, the bottleneck in comparing solutions by ISMAUT is that many instances of linear programming problems need to be solved. This added complexity weakens the efficiency of ISMAUT. We present an approach that avoids solving linear programming problems and hence greatly improve the efficiency of ISMAUT.

## Simulated Annealing Algorithm

This is an iterative improvement algorithms based on neighborhood search are widely used for hardware/software partitioning. To avoid being trapped in a local minimum heuristics are implemented which are very often based on simulated annealing. Simulated annealing selects the neighboring solution randomly and always accepts an improved solution. It also accepts worse solutions with a certain probability that depends on the deterioration of the cost function and on a control parameter called temperature. Simulated annealing algorithms can be quickly implemented and are widely applicable to many different problems. But it has some limitations such as long execution time and large amount of experiments needed to tune the algorithm. In Figure 3 we give a short description of the algorithm. With  $x$  we denote one solution consisting of the two sets  $Hw$  and  $Sw$ .  $x^{now}$  represents the current solution and  $N(x^{now})$  denotes the neighborhood of  $x$  now in the solution space.

```

Step 1.  Construct initial configuration  $x^{now} := (Hw_0, Sw_0)$ 
Step 2.  Initialize Temperature  $T := TI$ 
Step 3.  3.1. for  $i := 1$  to  $TL$  do
           Generate randomly a neighboring solution  $x' \in N(x^{now})$ 
           Compute change of cost function  $\Delta C := C(x') - C(x^{now})$ 
           if  $\Delta C \leq 0$  then  $x^{now} := x'$ 
           else
               Generate  $q := random(0, 1)$ 
               if  $q < e^{-\Delta C/T}$  then  $x^{now} := x'$ 
           3.2. Set new temperature  $T := \alpha * T$ 
Step 4.  if stopping criterium not met then goto Step 3
Step 5.  return solution corresponding to the minimum cost function

```

Fig. 3. Simulated annealing algorithm

A problem specific component of the SA algorithm is the generation of a new solution  $x'$  starting from the current one  $x^{now}$ . We implemented two strategies for solution generation: the *simple move* and the *improved move*. For the *simple move* a node is randomly selected for being moved to the other partition. The configuration resulted after this move becomes the candidate solution  $x'$ . Random node selection is repeated if transfer of the selected node violates some design constraints. The *improved move* accelerates convergence by moving together with the randomly selected node also some of its direct neighbors (nodes which are in the same partition and are directly connected to it). A direct neighbor is moved together with the selected node if this movement improves the cost function and does not violate any constraint. This strategy stimulates transfer of connected node groups instead of individual nodes. Experiments revealed a negative side effect of this strategy: the repeated move of the same or similar node groups from one partition to the other, which resulted in a reduction of the spectrum of visited solutions. To produce an optimal exploration of the solution space we combined movement of node groups with that of individual nodes: nodes are moved in groups with a certain probability  $p$ . After analysis of experimental results the value for  $p$  was fixed at 0.75. In Figure 4 we present the algorithm for generation of a candidate solution according to the improved move.

```

Step 1.  Generate  $k := \text{random}(1, \text{Nr\_of\_nodes})$  until moving node  $k$  does not violate constraints
Step 2.  Generate solution  $x'$  from  $x^{now}$  by moving node  $k$ 
Step 3.  Generate  $q := \text{random}(0, 1)$ 
Step 4.  if  $q \geq p$  then return  $x'$  and exit
Step 5.  5.1. for each node  $k'$  which was direct neighbor of node  $k$  and in the same partition with node  $k$  do
           if moving node  $k'$  does not violate constraints and produces  $\Delta C \leq 0$  then
               Generate solution  $x'$  from current  $x'$  by moving node  $k'$ 
           5.2. return  $x'$ 

```

Fig. 4. Generation of a new solution with improved move

## Tabu Search Algorithm

Tabu search controls uphill moves not purely randomly but in an intelligent way. The tabu search approach accepts uphill moves and stimulates convergence toward a global optimum by creating and exploiting data structures to take advantage of the search history at selection of the next move. Two key elements of the TS algorithm are the data structures called short and long term memory. Short term memory stores information relative to the most recent history of the search. It is used in order to avoid cycling that could occur if a certain move returns to a recently visited solution. Long term memory, on the other side, stores information on the global evolution of the algorithm. These are typically frequency measures relative to the occurrence of a certain event. They can be applied to perform *diversification* which is meant to improve exploration of the solution space by broadening the spectrum of visited solutions.

```

Step 1.  Construct initial configuration  $x^{now} := (Hw_0, Sw_0)$ 
Step 2.  for each solution  $x_k \in N(x^{now})$  do
        Compute change of cost function  $\Delta C_k := C(x_k) - C(x^{now})$ 
Step 3.  3.1. for each  $\Delta C_k < 0$ , in increasing order of  $\Delta C_k$  do
        if not  $tabu(x_k)$  or  $tabu\_aspirated(x_k)$  then
             $x^{now} := x_k$ 
            goto Step 4
        3.2. for each solution  $x_k \in N(x^{now})$  do Compute  $\Delta C_k := \Delta C_k + penalty(x_k)$ 
        3.3. for each  $\Delta C_k$  in increasing order of  $\Delta C_k$  do
        if not  $tabu(x_k)$  then
             $x^{now} := x_k$ 
            goto Step 4
        3.4. Generate  $x^{now}$  by performing the least tabu move
Step 4.  4.1. if iterations since previous best solution  $< Nr\_f\_b$  then goto Step 2
        4.2. if restarts  $< Nr\_r$  then
            Generate initial configuration  $x^{now}$  considering frequencies
            goto Step 2
Step 5.  return solution corresponding to the minimum cost function

```

Fig. 5. Tabu search algorithm

In Figure 5 we give a brief description of the TS algorithm. In a first attempt an improving move is tried. If no such move exists (or it is tabu and not aspirated) frequency based penalties are applied to the cost function and the best possible non tabu move is performed; this move can be an uphill step. Finally, in a last attempt, the move which is closest to leave the tabu state is executed. We consider as a candidate solution  $x_k$  the configuration obtained from  $x^{now}$  by moving node  $k$  from its current partition to the other one, if this movement does not violate any constraints. In the *tabu list* we store the list of the reverse moves of the last  $\tau$  moves performed, which are considered as being forbidden (tabu). The size  $\tau$  of this list (the *tabu tenure*) is an essential parameter of the algorithm. In Table 1 we present the optimal values for  $\tau$  as resulted from the experiments.

The total number of iterations performed for partitioning is influenced by parameters  $Nr\_f\_b$  (number of iterations without improvement of the solution after which the system is considered frozen) and  $Nr\_r$  (number of restarts with a new initial configuration). The minimal values needed for an optimal partitioning of all graphs of the respective dimension and the resulted CPU times are presented in Table 1. The times have been computed as the average of the partitioning time for all graphs of the given dimension. It is interesting to mention that restarting tours were necessary only for the 400 nodes graphs.

Parameters and CPU time with TS

| numbers of nodes | $\tau$ | Nr_f_b | Nr_r | CPU time (s) |
|------------------|--------|--------|------|--------------|
| 20               | 7      | 30     | 0    | 0.008        |
| 40               | 7      | 50     | 0    | 0.04         |
| 100              | 7      | 50     | 0    | 0.19         |
| 400              | 18     | 850    | 2    | 30.5         |

TABLE 1

### Energy-Conscious HW/SW Partitioning:

Energy dissipation is a hot topic in the design of – especially mobile - embedded systems. This is because applications like digital video cameras, cellular phones etc. draw their current from batteries that spend a limited amount of energy only. we show that energy-conscious HW/SW partitioning can lead to drastic reductions of energy dissipation of a whole embedded system. The obtained results show energy savings up 59% while the performance remains approximately the same or becomes even slightly higher. As a main result, energy-conscious HW/SW-partitioning is a promising method to be deployed in addition to classical energy and/or power reduction methods.

### Preference driven Hierarchical Approach

#### Hierarchical Models And the Approach

The behavior of an embedded system is usually represented by a task graph. A task graph is a directed acyclic graph in which each node represents a task and each edge represents the data dependency between the tasks. As pointed out in the introduction, it is often desirable to allow a hierarchical representation of tasks. That is, a complex task may consist of several simple tasks, and a simple task can contain some even simpler tasks. To facilitate such a system composition, we adopt a multiple granularity representation, called a *hierarchical task graph* (HTG).

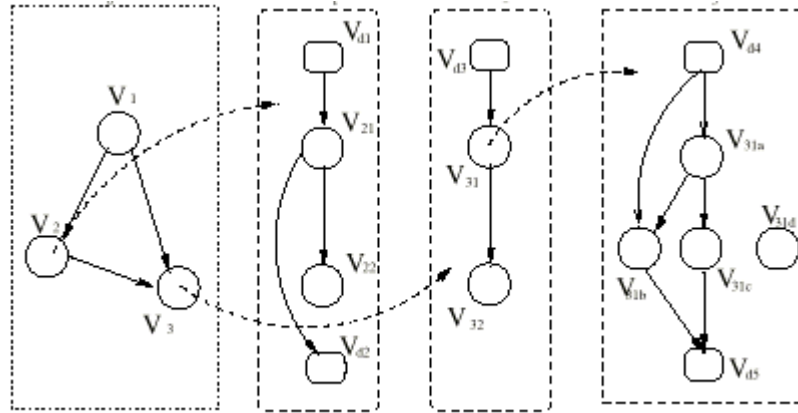


Figure 6. Hierarchical task graphs

An HTG is a task graph which contains three kinds of nodes: simple nodes, complex nodes and dummy nodes. A simple node is a node representing a task containing no sub-tasks. The task represented by a complex node can be decomposed to several sub-tasks, which can be expressed in more detail by another lower level HTGs, i.e. sub-HTGs. The third kind of nodes are dummy nodes, which exist in sub-HTGs. A dummy node represents only the input and output relation with other HTGs and is not associated with any computational task. The behavior of a system as well as its hierarchical structure can be represented by a set of HTGs. Figure 1 depicts an HTG example. In Figure 6,  $G_0$  is the highest level HTG representing a system with one simple task (corresponding to  $V_1$ ) and two complex tasks (corresponding to  $V_2$  and  $V_3$ ). The two complex tasks are represented further by sub-HTGs  $G_1$ ,  $G_2$ , and  $G_3$ . Specifically in  $G_2$ ,  $V_{32}$  is a simple node,  $V_{31}$  is a complex node,  $V_{d3}$  and is a dummy node.

Though an HTG is a convenient representation to capture the intrinsic hierarchical structure of the functionality for an embedded system, it does not represent a *complete system* behavior at different hierarchical levels (except for the highest one). In order to clearly differentiate behavior models of a complete system at various level of hierarchy, the concept of *HTG instance* has been introduced. An HTG instance is a task graph that combines appropriate HTGs in different levels to describe the behavior of the whole system. For example, in Figure 6, by replacing the complex nodes  $V_2$  and  $V_3$  in HTG  $G_0$  with their sub-HTGs,  $G_1$  and  $G_2$ , we can construct a new task graph shown in Figure 7(a). The task graph in Figure 7(a) describes the same system behavior as HTG  $G_0$  in Figure 1 but in more detail. The task graph in Figure 7(b) is another instance which is constructed from  $G_0$ ,  $G_1$ ,  $G_2$  and  $G_3$ . Given the set of HTGs for a system, by expanding different complex nodes, we can construct different HTG instances with different granularities.

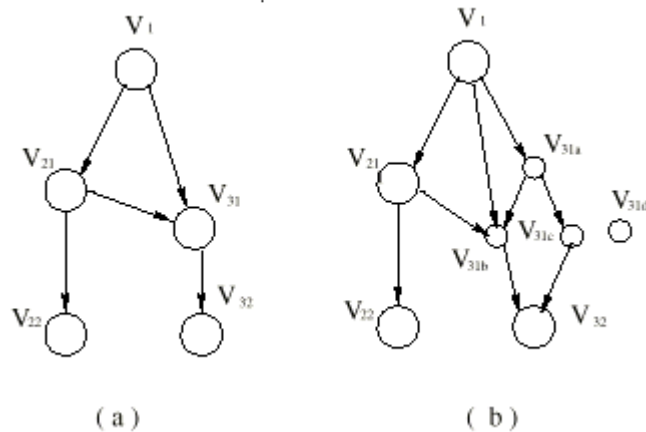


Figure 7. Hierarchical Task Graph instances

We consider *hardware objects* to be individual chips, IP components, or functional components such as processors, ASICs, and programmable devices. A *hardware module* is an instance of a hardware object. We adopt a hierarchical representation for hardware modules in a similar way as that for the system behavioral specification. Such a hierarchical structure

for hardware objects is quite natural. For example, a coarse-grain hardware may represent a chip which contains a CPU core and peripheral circuitry. The peripheral circuitry, in turn, consists of modules at an even lower level such as timing circuits, A/D and D/A converters and so on. Such hierarchical information can be readily captured by the tree representation.

We consider the following hardware/software partitioning problem: given (1) a set of HTGs, which describe the behavior of an embedded system, (2) hierarchical hardware modules, (3) communication links, (4) constraints on cost, power, timing, etc., and (5) a designer's preference for certain attributes, find an assignment of task nodes of one HTG instance to some hardware modules in a way that optimizes all design attributes while satisfying all design constraints. Considering the NP-hard nature of the partitioning problem, we use an evolutionary algorithm (EA) to search for high quality solutions. EAs have been widely used in a wide variety of optimization problems. EAs work well for solving the non-hierarchical partitioning problem for embedded system. If the system task specifications as well as hardware modules contain hierarchical structures, simply employing an EA—or any other type of search algorithm—may not lead to an efficient search process. Either the computational overhead may prove to be too costly or one has to settle for an inferior solution. Nonetheless, EAs are well suited for solving partitioning problems.

## Hierarchical Evolutionary Algorithm

In order to utilize the hierarchical structure for efficient exploration of the design space, we propose a hierarchical-structure based EA. In virtually all implementations of EAs, the size of a genotype i.e., the data structure which encodes a solution—is fixed *a priori*. Unfortunately, a fixed size genotype cannot readily handle the hierarchical structure in the partitioning problem because both simple and complex nodes must be accommodated. Moreover, the reproduction operators must likewise be dynamically changed as the genotype size changes. We refer to such an EA as *hierarchical EA* (HEA). In the following, we discuss in more detail the data structure and reproduction strategy in a HEA.

### Data Structure

In hardware/software partitioning problem, for a non-hierarchical task graph, each node is to be assigned to a hardware module. In EA, such a node-hardware tuple becomes a gene in an individual. However, for the hierarchical task graph, how to encode genes needs some careful consideration. A simple approach is to associate each element with a finest level task node. This approach maintains the same number of elements in all individuals but diminishes the advantage of the hierarchical representation. Another intuitive approach is to associate each element with either a complex or simple node in HTGs. The problem with this approach is that a basic task may be represented implicitly more than once.

Additional effort would be required during the construction of solutions to avoid any undesirable conflicts for this approach. Recall that an HTG instance itself is a task graph that represents the complete system behavior. Hence, we construct individuals from the HTG instances instead of HTGs. Each individual is related to one HTG instance of the given HTGs, and each gene in the individual corresponds to a node in the HTG instance. Note that no task is represented more than once in an HTG instance. This guarantees the correctness when constructing the individual. We use the notation  $(V_i; M_k)$  to denote task  $V_i$

is assigned to (hardware) module  $M_k$ . Then a *gene list* for the instance in Figure 2(a) might be  $\{(V_1, M_1) (V_{21}, M_2) (V_{22}, M_3) (V_{31}, M_4) (V_{32}, M_5) \}$  and  $\{(V_1, M_1') (V_{21}, M_2) (V_{22}, M_3') (V_{31a}, M_4') (V_{31b}, M_5') (V_{31c}, M_6') (V_{31d}, M_7') (V_{32}, M_8') \}$  for Figure 2(b). Note that this notation naturally reflects different granularity levels, which is necessary because distinct individuals may have different sizes.

In a HEA the nodes are assigned in a unique order and the genes are listed by the order of their corresponding nodes. However, in HEA the order is defined in *each* HTG. Sub-node inherits the order of its parent. Specifically, given two nodes  $u$  and  $v$  in the same HTG (neither  $u$  nor  $v$  is a parents of the other), assume that  $u$  precedes  $v$  according to the given order. We denote this by  $O(u) \rightarrow O(v)$ . Suppose that  $u$  is a complex node, then the nodes in the corresponding sub-task graphs  $G(u) = (V_u, E_u)$  must satisfy  $O(x) \rightarrow O(v)$ ,  $x$  belongs to  $V_u$ . Later on, we will show the importance that order plays in maintaining the consistency of tasks and modules when complex nodes are expanded.

## Reproduction

In a HEA, the reproduction process stochastically creates new (offspring) solutions from existing (parent) solutions. Because the size of the gene lists changes dynamically in HEA, careful design of reproduction operators is critical to achieve efficient design space exploration. In HEA, both mutation and crossover are designed to generate individuals with different granularities as well as same granularity.

Mutation generates new species by updating one gene of an individual. There are two ways this can be done: change the hardware module to which the task is assigned in the gene or, if the node is complex, replace the gene with the genes associated with the sub-node set of the complex node. The advantage of HEA is that evolution can be first performed at coarse grain level, which explores only a relatively small search space. Only when it seems to be difficult to satisfy the constraints of the system, there is a need for exploring alternatives at the finer levels (a larger search space). Thus the search space is maintained as small as possible to improve the search effectiveness and efficiency.

The replacement of a complex node with its sub-node set is done with probability  $\Theta$ , and the mapping of  $V_i$  to another randomly selected hardware module is done with probability  $1 - \Theta$ . The value of  $\Theta$  increases with the number of generations (iterations) during which the individual with the highest fitness value (solution quality) has not been improved. Intuitively, the value of  $\Theta$  is larger at higher levels in the hierarchy because the search space at the higher level is much smaller. We use the following formula to calculate  $\Theta$  in our system,

$$\Theta = (1 - \frac{1}{g}) \times \frac{N_s - N_i}{N_s} + k$$

where  $g$  is the generations when the highest fitness value has not been improved,  $N_s$  is the total number of simple nodes in the finest level of HTG,  $N_i$  is the number of the nodes in the current HTG instance, and  $k$  is a constant defined by the user. Consider the

example in Figure 2(a). We have  $N_s = 8$  and  $N_i = 5$ . Let  $k = 0.1$ , then if the best solution has not been improved after 5 generations, we have  $\Theta = 0.40$ .

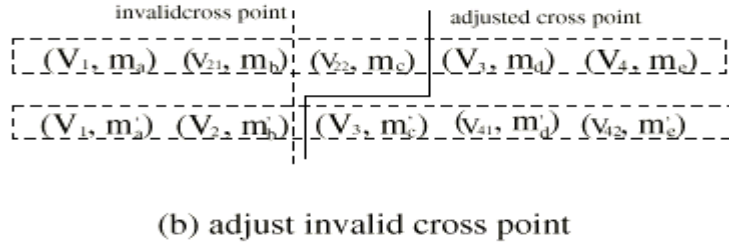
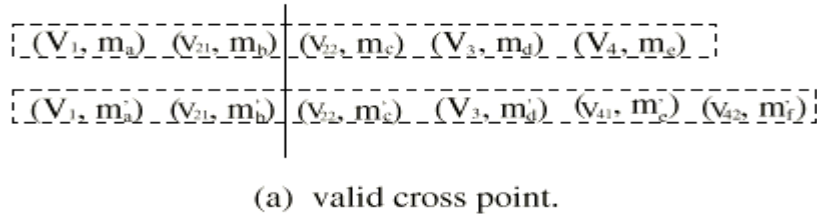


Figure 8. Crossover in HEA

The crossover points must be chosen with care because the parent individuals may potentially have different granularities. Specifically, it is very important to guarantee that the newly created individuals are valid solutions. As shown in Figure 8(a), when two gene lists are sliced along the crossover point, the parts on the same side of crossover point are associated with the same functionality, which makes such a crossover point valid. Otherwise, as shown in Figure 8(b), simply exchanging the subset of genes of both parents would generate invalid individuals. To overcome this problem, we need to adjust the position of crossover point such that the cross line can cut along the “boundary” of higher level task in both individuals. Note that the data structure for the individuals maintains a total order for tasks as defined earlier. Therefore, finding the correct position for the new crossover point simply requires linear scanning the genes lists. After the correct crossover point is identified, we can swap the two subsets of two parents to obtain two new offsprings.

There are several other issues needed more considerations in HEA, such as hardware module consistency (a complex module may appear in the same solution with its sub-modules), attribute calculation, scheduling, and so on.

## Preference Driven Ranking

When solving the partitioning problem, how to handle the multiple, often conflicting design objectives is not easy. ISMAUT offers an efficient way to compare alternative design according to the designer’s preferences. For completeness, we briefly review the ISMAUT approach. ISMAUT uses a linear weighted-sum format to capture the fitness of a design alternative. Let the fitness of a design  $x$  be represented by  $V_x$ , and denote the  $k$ th of  $x$  attribute by  $a_k(x)$ , then

$$V_x = \sum_k w_k \times v_k(a_k(x))$$

where  $v_k(\cdot)$  maps the raw attribute values to set  $[0,1]$  and  $w_k$  is the corresponding weight. A bigger value of  $V_x$  indicates a more desirable design alternative.  $V_x$  is imprecisely defined in the sense that each  $w_k$  does not have a specific value, but is constrained by the designer's preferences as follows.

The designer's preferences are indications of which attributes are considered to be more important and which designs are considered to be more desirable. Let  $x, x'$  be two individuals with attribute  $a_k(x)$  and  $a'_k(x)$ ,  $k = 1, 2, \dots, n$ . Suppose that according to the designer preference,  $x$  is considered to be preferable to  $x'$ , denoted by  $x > x'$ . We can derive one constraint for  $W$  as:

$$V_x - V_{x'} = \sum_k w_k [v_k(a_k(x)) - v_k(a_k(x'))] > 0$$

When more than one pair of design alternatives are ranked by the designer, a set of such constraints are defined which confines  $W = (w_1, w_2, \dots, w_n)$  to a subspace of  $W_n$ . With these constraints, any two design alternatives, i.e.  $y$  and  $y'$  can be compared by solving the following two linear programming problems,

$$\text{Min} : \mu = \sum_k w_k [v_k(a_k(y')) - v_k(a_k(y))]$$

$$\text{s.t.} : \overline{W} = (w_1, w_2, \dots, w_n) \in W^n$$

and

$$\text{Min} : \mu' = \sum_k w_k [v_k(a_k(y)) - v_k(a_k(y'))]$$

$$\text{s.t.} : \overline{W} = (w_1, w_2, \dots, w_n) \in W^n$$

If  $\mu > 0$  (or  $\mu' > 0$ ), then  $x' > x$  (or  $x > x'$ ). If  $\mu < 0$  and  $\mu' < 0$ , then  $x'$  and  $x$  are indifferent ( $x' \sim x$ ), i.e., no one is clearly better than the other. Notice that a large number of linear programming instances must be solved in order to rank many design solutions which is very time consuming when a large number of individuals need to be compared in many generations in EA.

However, once the designer's preferences have been given, the constraint space  $W_n$  is fixed. This makes it possible to avoid solving the linear programming problems one by one. In fact, the minimum values of the objective function for the linear programming problems are always attained at one of the extreme points defined by the set of constraint.

Solving the linear programming problems can therefore be transformed to check the objective function values at each of these extreme points.

| Preference<br>Constraint | Dimension of $W^n$ |     |     |     |     |
|--------------------------|--------------------|-----|-----|-----|-----|
|                          | 10                 | 25  | 50  | 75  | 100 |
| 10                       | 9                  | 13  | 20  | 23  | 35  |
| 20                       | 28                 | 75  | 80  | 62  | 110 |
| 30                       | 30                 | 108 | 292 | 280 | 242 |
| 40                       | 49                 | 218 | 252 | 328 | 611 |
| 50                       | 46                 | 313 | 543 | 582 | 793 |

Table 2. Efficient Extreme Points

To find the extreme points from the given designer's preference, we make use of a software package called ADBASE. Each extreme point in our case corresponds to a vector in  $W_n$ . A pair of indifferent individuals,  $x$  and  $y$ , can thus be compared by evaluating the following values with respect to the extreme point set.

$$\min(V_x - V_y) = \min\left(\sum_k w_k [v_k(a_k(x)) - v_k(a_k(y))]\right)$$

One concern about using this approach is that the number of the extreme points grows rapidly with the dimension of  $W_n$ , and the number of preference constraints. However, this approach is still more efficient than that of directly solving linear programming problems. Table 1 illustrates the relationship between constraints and dimension. Consider the case of a design alternative with 10 attributes and 20 preference constraints. Referring to Table 2, we have at most 28 extreme points. Let  $t_e$  be the amount of time needed to calculate the extreme points with ADBASE, and  $t_0$  be the time to compute the fitness value at each extreme point (which is the amount of time for performing 10 multiplications and 9 additions), and  $t_1$  be the amount of time for comparing two fitness values. Then in the worse case, the total amount of time  $T_A$  needed to rank 100 individuals in 100 generations can be obtained by

$$T_A = t_e + 100 \times (100 \times 28 \times t_0 + (100 \times 99/2) \times t_1)$$

If the ISMAUT approach is used for the above example, in the worse case, the total amount of time  $T_{LP}$  would be

$$T_{LP} = 100 \times (100 \times 99 \times t_{LP})$$

where  $t_{LP}$  is the amount of time required to solve a linear programming problem with 10 variables and 20 constraints. In most cases,  $t_e \approx t_{LP}$ ,  $t_{LP} \gg t_0$ , and  $t_{LP} \gg t_1$ , so  $T_{LP} \gg T_A$ . Our experimental results also agree with this conclusion.

## Future possibilities

1. Many systems exhibit a high degree of regularity (regularity means many of the behaviors in the system are identical, differing only in the data on which they operate). Future algorithms should include techniques to partition regular and semi-regular behaviors such as feedback the metrics to the system and partition again.
2. Since partitioning is a quite mature field, the majority of future tasks will involve adaptation of existing techniques for applicability at the functional level.
3. Develop an algorithm which partition at multiple levels of granularity.
4. Combine functional partitioning with high-level synthesis.

## Conclusion

In this paper, we discussed various issues related to HW/SW partitioning and presented several techniques to improve the hardware/software partitioning process for large, complex embedded systems. We described the use of both hierarchical task specification and hardware modules. To facilitate the partitioning process, the existing EA approach was extended so that it can effectively handle hierarchical structures. To further improve the efficiency of the preference-driven hardware/software partitioning process, we use the idea of employing the extreme points in multi-objective linear programming to eliminate the time-consuming procedure of solving multiple linear-programming problem instances.

## References

- ◆ Petru Eles, Zebo Peng, Krzysztof Kuchcinski, Alexa Doboli “*System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search*”
- ◆ Gang Quan Xiaobo(Sharon) Hu Garrison Greenwood “*Preference-Driven Hierarchical Hardware/Software Partitioning*”
- ◆ Jörg Henkel Yanbing Li “*Energy-Conscious HW/SW-Partitioning of Embedded Systems: A Case Study on an MPEG-2 Encoder*”
- ◆ Gajski D D, Vahid F, Narayan S, Gong J “*Specification and Design of Embedded Systems*”