Preference-Driven Hierarchical Hardware/Software Partitioning *

Gang Quan Xiaobo(Sharon) Hu
Dept. of Computer Science & Engineering
University of Notre Dame
Notre Dame, IN 46556
{gquan,shu}@cse.nd.edu

Garrison Greenwood
Dept. of Electrical & Computer Engineering
Western Michigan University
Kalamazoo, MI 49008
garry.greenwood@wmich.edu

Abstract

In this paper, we present a hierarchical evolutionary approach to hardware/software partitioning for real-time embedded systems. In contrast to most of previous approaches, 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. Two new search operators are described, which exploit the proposed hierarchical structure. Efficient ranking is another problem addressed in this paper. 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 propose a new technique to reduce the overhead. Experiment results show that our algorithm is both effective and efficient.

1. Introduction

Hardware/software partitioning, is an important step in hardware/software codesign that determines which system tasks should be realized in which hardware modules. 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 [14].

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 [3–5, 12]. 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. We believe 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 are easier to deal with. Researchers have adopted hierarchy for hardware/software partitioning problems in a variety of ways [2,6]. However, we feel the hierarchal structure in the embedded system can be exploited more efficiently and aggressively. In this paper, we propose a new partitioning methodology which incorporates hierarchical structures for both tasks and hardware modules. We employ an evolutionary algorithm (EA) to efficiently the handle hierarchical tasks and hardware modules. In our 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 ex-

*This research is supported in part by an External Research Program Grant from Hewlett-Packard Laboratories, Bristol, England, by DARPA/Army under contract number DABT63-97-C-0048, and by NSF under grant number MIP-9796162 and MIP-9701416.

ample, 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 [3], is based on the assumption that all Pareto optimal solutions are equally preferable—a situation that does not hold in many real world applications. In [13, 15], the *Imprecisely Specified Multiple Attribute Utility Theory* (ISMAUT) is used, which combines the advantages of both a weighted-sum method and a Pareto optimal ranking algorithm. With ISMAUT, comparsion 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. In this paper, we present an approach that avoids solving linear programming problems and hence greatly improve the efficiency of ISMAUT.

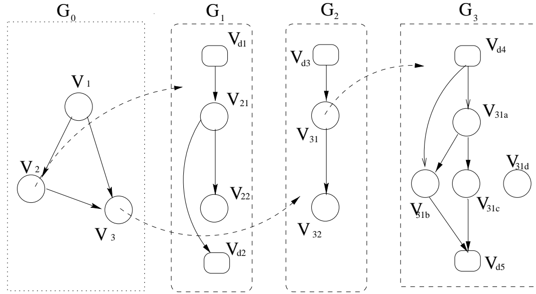


Figure 1. Hierarchical task graphs

The paper is organized as following. Section 2 describes our problem in more detail. Section 3 describes the hierarchical evolutionary algorithm. Section 4 describes the preference driven technique. Finally, we present some experiment results of applying our algorithm.

2. Hierarchical Models And Our Approach

The behavior of an embedded system is usually represented by a task graph [4, 14]. 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).

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 1, 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 furthur 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, and V_{d3} 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, we introduce the concept of *HTG instance*. 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 1, 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 2(a). The task graph in Figure 2(a) describes the same system behavior as HTG G_0 in Figure 1 but in more detail. The task graph in Figure 2(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.

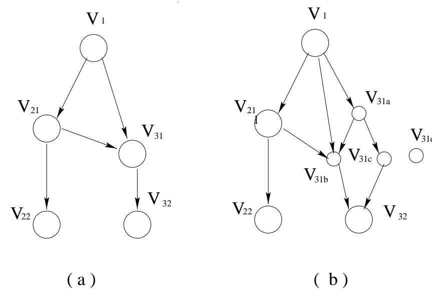


Figure 2. Hierarchical Task Graph instances

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 exam-

ple, 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 been widely used in a wide variety of optimization problems [1]. EAs work well for solving the non-hierarchical partitioning problem for embedded system. In fact, applying EAs to the partitioning problems has been investigated in several papers [3,5,7,12,15]. However, 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, our studies do indicate that EAs are well suited for solving partitioning problems.

3. 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.

3.1. 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 or v is a parents of the other), assume that u preceeds v according to the given order. We denote this by $O(u) \mapsto O(v)$. Suppose that u is a complex node, then the nodes in the corresponding sub-task graphs $G(u) = (V_u, E_u)$ must satisfy $O(x) \mapsto O(v),\ x \in V_u$. Later on, we will show the importance that order plays in maintaining the consistency of tasks and modules when complex nodes are expanded.

3.2. 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.

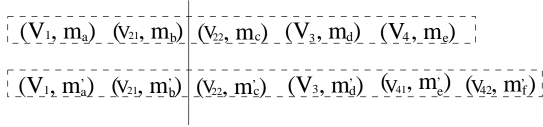
Mutation generates new species by updating one gene of an individual. There are two ways this can be done: change the hardware module to which the task is assigned in the gene or, if the node is complex, replace the gene with the genes associated with the sub-node set of the complex node. The advantage of HEA is that evolution can be first performed at coarse grain level, which explores only a rel-

atively 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.

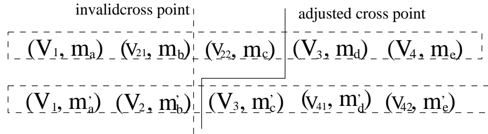
The replacement of a complex node with its sub-node set is done with probability Θ, and the mapping of V_i to another randomly selected hardware module is done with probability $1-\Theta$. The value of Θ increases with the number of generations (iterations) during which the individual with the highest fitness value (solution quality) has not been improved. Intuitively, the value of Θ 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 Θ 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 $\Theta = 0.40$.



(a) valid cross point.



(b) adjust invalid cross point

Figure 3. 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 3(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 fuctionality, which makes such a crossover point valid. Otherwise, as shown in Figure 3(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 in Section 3.1. 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 submodules), attribute calculation, scheduling, and so on. Due to the page limit, we omit the detailed explaination. Interested readers can refer to [9] for more information.

4. Preference Driven Ranking

When solving the partitioning problem, how to handle multiple, often conflicting design objectives is not easy. ISMAUT offers an efficient way to compare alternative design according to the designer's preferences. Details on ISMAUT can be found in [13, 15]. 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 kth 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 $a'_k(x)$, $k = 1, 2, \cdots, n$. Suppose that according to the designer preference, x is considered to be preferable to x', denoted by $x \succ x'$. We can derive one constraint for $\overline{W}$ 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 $\overline{W} = (w_1, w_2, \cdots, w_n)$ 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 lin-

ear programming problems,

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

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

and

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

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

If $\mu > 0$ (or $\mu' > 0$), then $x' \succ x$ (or $x \succ x'$). 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 [11]. Solving the linear programming problems can therefore be transformed to check the objective function values at each of these extreme points.

Preference 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 1. Efficient Extreme Points

To find the extreme points from the given designer's preference, we make use of a software package called ADBASE [10]. 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.

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

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. In [10], the relation between the number of extreme points, dimension and number of constraints was studied.

Table 1 was copied from [10] to illustrate the relationship between constraints and dimension. Consider the case of a design alternative with 10 attributes and 20 preference constraints. Referring to Table 1, 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

$$T_A = t_e + 100 \times (100 \times 28 \times t_0 + \frac{100 \times 99}{2} \times t_1)$$

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, $t_e \simeq t_{LP}$, $t_{LP} \gg t_0$, and $t_{LP} \gg t_1$, so $T_{LP} \gg T_A$. Our experimental results also agree with this conclusion.

5. Experimental Results

We have implemented the ideas in this paper in a software package called **EvoC**. The input of **EvoC** consists of the HTGs, task attributes, data dependency, hierarchical information, hardware modules, etc. The output of **EvoC** contains the task assignment, dollar cost, average power consumption, and the task execution schedule. Because of its simplicity, the list scheduling algorithm [8] is used in **EvoC**.

In our examples, we use the task graphs and parameters including task execution times, deadlines, communication data volume, and hardware module information in [4]. Hierarchical information is derived from the clustered and unclustered task graphs in [4]. We modify the period of the task graph to be the latest deadline in the task graph. This modification simplifies the scheduling process as scheduling is not our focus in this paper. Power consumption data is obtainted from the FTP site ftp://ftp.ee.princeton.edu/pub/dickrp/Trans/Mogac [3]. All results were obtained on a SUN Ultra-1 workstation.

Table 2 contains experiments on two sets of tasks chosen from the four task graphs in [4]. The first row is for the hierarchical approach. The second row shows the results

Task Set	Set 1 (task1&2)						Set 2 (task3&4)					
	Initial		Result				Initial		Result			
Attribute	**Cost**	**Power**	**Cost**	**Power**	**Gen.**	**CPU(s)**	**Cost**	**Power**	**Cost**	**Power**	**Gen.**	**CPU(s)**
Hier.	170	50.6	100	36.3	12	3.45	190	59.3	170	38.3	15	9.45
Uncluster	190	49.3	170	34.3	39	9.05	190	70.2	170	38.5	57	56.84
Cluster	170	44.3	100	36.3	15	3.25	170	56.3	170	43.0	9	1.53
PrefADBASE	190	49.3	100	36.3	69	32.86	190	70.2	170	38.5	51	106.3
PrefISMAUT	190	49.3	100	36.3	69	71.32	190	70.2	170	38.5	51	356.6

Table 2. Hardware/software partitioning examples

of partitioning at a finer granularity (10 tasks in each task graph). The third row shows the results of partitioning at a coarse granularity (3 or 4 tasks in each task graph). From Table 2, with the hierarchical approach, we can get results comparable in quality with those by searching at the finest level (Uncluster) and much better than those by searching at coarse grain level (Cluster). For CPU time, the hierarchical approach takes little more than that of the coarse grain search but much less than that of the finest grain search. The effectiveness and efficiency of hierarchical approach is apparent.

While dominance checking is used when comparing two individuals for the first three rows, the last two rows show the partition results of an unclustered task graph with designer's preferences. ADBASE was used for the results in the fourth row, while ISMAUT is used for those in the fifth row. Compared with solving linear programming problems in ranking alternatives, the speedup by using ADBASE is more than 2 (3) times that of set 1 (set 2). This confirms our analysis from Section 4.

6. Summary

In this paper, we present several techniques to improve the hardware/software partitioning process for large, complex embedded systems. We proposed the use of both hierarchical task specification and hardware modules. To facilitate the partitioning process, we extended the existing EA approach so that it can effectively handle hierarchical structures. To further improve the efficiency of the preference-driven hardware/software partitioning process, we introduced 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. The experimental results obtained so far have clearly demonstrated the advantages of our proposed approach.

References

[1] T. Bäck, U. Hammel, and H. P. Schwefel. Evolutionary computation: Comments on the history and current state. *IEEE Trans. on Evolution. Comp.*, 1(1):3–17, 1997.

[2] B. Dave and N. Jha. Cohra:hardware-software co-synthesis of hierarchical distributed embedded system architectures. *The 11th International Conf. on VLSI Des.*, pages 347–354, 1998.

[3] R. Dick and J. Jha. Mogac:a multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems. *IEEE Trans. on CAD of Integr. Circ. & Sys.*, 17(10):920–935, 1998.

[4] J. Hou and W. Wolf. Process partitioning for distributed embedded systems. *IEEE/AMC Intl. Workshop on Hardware-Software Codesign*, 1996.

[5] X. Hu and G. Greenwood. Evolutionary approach to hardware/software partitioning. *IEE Proc. on Comput. Digit. Tech.*, 145(3):203–209, 1998.

[6] J. Kenkel and R. Ernst. A hardware/software partitioner using a dynamically determined granularity. *IEEE Intl. Conf. on Design Automation Conf.*, pages 691–696, 1997.

[7] Y. Kwok and I. Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *Parallel and Distributed Computing*, 47:58–77, 1997.

[8] E. Lawler and C. Martel. Scheduling periodically occurring tasks on multiple processors. *Info. Proc. Ltr.*, 7:9–12, 1981.

[9] G. Quan, X. Hu, and G. Greenwood. Hierarchical hardware/software partitioning. *Technological Reports, Department of Computer Science and Engineering, University of Notre Dame*, (TR 99-11), 1999.

[10] R. Steuer. Manual for the adbase multiple objective linear programming package. 1995.

[11] J. Strayer. Linear programming and its applications. *NewYork: Springer-Verlag*, 1989.

[12] J. Teich, T. Blickle, and L. Thiele. An evolutionary approach to system-level synthesis. *Proc. of Fifth Int'l Workshop on Hardware/Software Codesign*, 1997.

[13] C. White, A. Herraya, and S. Dozono. A model of multi-attribute decision making and trade-off weight determination under uncertainty. *IEEE Trans. on Sys, Man, and Cyber.*, SMC-14(2):223–229, 1984.

[14] W. H. Wolf. Hardware-software co-design of embedded systems. *Proc. of the IEEE*, 82(7):967–989, 1994.

[15] G. G. X. Hu and J. D'Ambrosio. An evolutionary approach to hardware/software partition. *The 4th Int'l Conf. on Parallel Problem Solving from Nature*, pages 900–909, 1996.