Handling Constraints in Multi-Objective GA for Embedded System Design

Biman Chakraborty Ting Chen Tulika Mitra Abhik Roychoudhury
National University of Singapore
stabc@nus.edu.sg, {chent,tulika,abhik}@comp.nus.edu.sg

Abstract

Design space exploration is central to embedded system design. Typically this is a multi-objective search problem, where performance, power, area etc. are the different optimization criteria, to find the Pareto-optimal points. Multi-objective Genetic Algorithms (GA) have been found to be a natural fit for such searches and have been used widely. However, for certain design spaces, a large part of the space being explored by GA may violate certain design constraints. In this paper, we use a multi-objective GA algorithm based on "repair", where an infeasible design point encountered during the search is repaired to a feasible design point. Our primary novelty is to use a multi-objective version of search algorithms, like branch and bound, as the repair strategy to optimize the objectives. We also precompute a layout of the genes such that infeasible design points are less likely to be encountered during the search. We have successfully employed our hybrid search strategy to design application-specific instruction-set extensions that maximize performance and minimize area.

1 Introduction

Embedded system design can be viewed as a complex design space exploration problem that attempts to optimize conflicting criteria, for example, maximizing performance while minimizing energy and area requirement. The multi-objective nature of these optimization problems implies that the search returns a set of Pareto-optimal design points. A design point p is Pareto-optimal if there does not exist any other design point p' that is superior to p in terms of all the objectives. Multi-objective Genetic Algorithms (GA) are quite suitable in discovering Pareto fronts. GA is much more efficient than exact algorithmic searches and even the non-exhaustive ones like branch and bound [1]. Therefore, several research groups have successfully applied multi-objective GA for embedded system design (see [6] for an overview of this topic). Most of these optimization problems, where GA has been hugely successful, are characterized by a design space that contains very few infeasible points.

In this paper, we show that there exists another class of multi-objective optimization problems in the context of embedded system design where a large fraction of the design space is infeasible (i.e., violates one or more design constraints). We argue that GA performs poorly for such problems as the mutation and the crossover operations result in a large percentage of the population being infeasible after a few generations. We then propose a hybrid optimization strategy that combines GA with branch and bound search. Our hybrid algorithm *repairs* an infeasible point generated by GA to a feasible point. Moreover, to improve the quality of the design points, we repair an infeasible point to a "neighboring" point that optimizes the objectives. This is accomplished via a *multi-objective version of branch and bound* — exact search and optimization over a small part of the design space. We also pre-compute a gene layout for GA such that crossover operation is less likely to produce infeasible design points.

To evaluate the effectiveness of our hybrid search strategy, we have employed it for the design of application-specific instruction-set extensions (*custom instructions*) [9]. Custom instructions encapsulate the frequently occurring computation patterns in an application and help simple embedded processors achieve considerable performance and energy efficiency. However, choosing an appropriate set of custom instructions for an application so as to maximize performance while minimizing area/energy is a difficult problem. In our experiments, we found that our combination of multi-objective GA and branch and bound for custom instructions design significantly improves the quality of the solutions.

Related Work Design space exploration of embedded systems is a well-studied problem. Apart from multi-objective GA, substantial research has been done in developing heuristic search techniques for computing the set of Pareto-optimal design points. Platune [8] and PICO (Program In Chip Out) [10] are some examples of heuristic multi-objective design space exploration frameworks. The

heuristic algorithms typically exploit dependence among the parameters, sensitivity of the objective function on the parameters [7], symbolic constraint satisfaction etc. to prune the design space. Design space exploration for custom instructions has also received lot of attention lately. Given a set of candidate patterns, various approaches have been proposed to select the optimal subset under different constraints. These include exhaustive search [2], ILP [13] and heuristic algorithms [3, 13]. To the best of our knowledge, ours is the first work that looks at the multi-objective selection problem for custom instructions. Our search technique handles constraint violations in multi-objective GA by repairing infeasible solutions. Quite a few constraint handling methods exist in the GA literature [11]. One of them is to preserve the feasibility of the solutions, that is, two feasible solutions, after crossover and mutation operations, are guaranteed to create two feasible offsprings. While this approach works well for some constrained problems, it very much depends on the specific problem at hand and the generalization of these redefined operators to other similar problems is by no means obvious. The most popular approach in handling the constraints is to use penalties. The basic idea is to define the fitness value of an infeasible solution with a penalty term. [4] discusses various issues related to the use of penalty functions. One major drawback of this approach is that it redefines the optimization problem and the optimal solution depends on the penalty parameters. Our strategy of repairing an infeasible solution raises another concern – the computational cost of repair. For the custom instruction problem we show that the time overhead for repair is well-compensated by the superiority of the solutions produced.

2 Difficult Multi-objective problems in Embedded System Design

A closer look at the constraints for embedded system optimization problems reveals that these constraints can be divided into two classes. The first class of constraints arises due to the limited resources such as area/energy. These constraints are replaced by objective functions in the multi-objective version of the problem. Thus, if the multi-objective optimization problem has performance and area as objectives, the corresponding single objective version of the problem may optimize performance subject to an area budget A, i.e., with the constraint $area \leq A$.

The second class of the constraints can be loosely termed as *mutual exclusion* constraints. A mutual exclusion constraint is of the form $V_1 + \ldots + V_n \leq 1$ where $V_1, \ldots, V_n$ are 0/1 integer decision variables. $V_1, \ldots, V_n$ typically represent the different choices with corresponding performance, energy, and area functions and at most one of these choices can be selected. Mutual exclusion constraints are quite prevalent in embedded design space exploration problems such as HW/SW partitioning, application-specific instructions selection, code optimization etc.

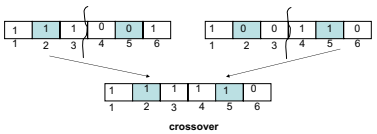


Figure 1. *Example of infeasible solution being constructed from feasible solutions by crossover.*

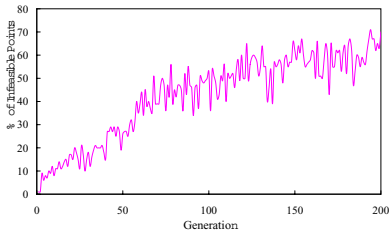


Figure 2. *Percentage of infeasible points over generations for custom instruction selection with* `djpeg`

The presence of mutual exclusion constraints creates difficulty for multi-objective GA. In GA, each binary decision variable is represented by a gene and the sequence of genes forms a chromosome representing a particular design point. The basic assumption of GA is that the values of the different genes are independent, i.e., there is no *epistasis* or interaction among the different genes. Unfortunately, the mutual exclusion constraints imply the presence of epistasis and not all design points are feasible. Both the classical crossover and mutation might not respect this constraint. In Figure 1, we show how this can happen via a crossover operation. In this example, there is only one mutual exclusion constraint — we cannot set both gene 2 and gene 5. The parent chromosomes both satisfy this constraint but the child chromosome produced by crossover shown in Figure 1 does not satisfy this constraint.

As a result of infeasible solutions being constructed (from feasible solutions) by crossover (and mutation too), GA may end up with significant fraction of infeasible design points in its population. For example, Figure 2 shows the percentage of infeasible points in the population of 100 chromosomes over 200 generations for custom instruction selection problem (benchmark `djpeg`). We make sure that all the design points in the initial population are feasible. Still, as the evolution progresses, we can observe as much as 70% infeasible points in a generation. This shows that we need to go beyond simple multi-objective GA methods.

3 Handling mutual exclusion constraints

In this section, we discuss novel techniques used in our work to handle mutual exclusion constraints in multi-objective GA. We assume familiarity with basic GA terminology, such as gene, chromosome, selection, crossover and mutation (see [5] for an overview).

3.1 Optimizing gene layout

To deal with mutual exclusion constraints, multi-objective GA should be careful about the choice of chromosome encoding. In the following, we use the words "chromosome" and "solution" interchangeably. The most straightforward encoding scheme is to *randomly* assign the genes in a chromosome to the binary design variables. Clearly in such an encoding scheme, design variables participating in the same mutual exclusion constraint may be spread throughout the chromosome. In contrast, if design variables participating in the same constraint are close together in the chromosome, they are less likely to be separated under a single-point crossover.

In problems with mutual exclusion constraints, we propose the following solution. We say that a gene is *covered* by a mutual exclusion constraint C if it is placed in the chromosome between (any) two genes participating in C. When we perform a single point cross-over at a randomly chosen gene, all the constraints covering the gene will be broken apart. The resultant offsprings will inherit the gene values of those broken constraints from different parents. Because only one gene in a mutual exclusion constraint is allowed to take the value 1, an offspring may become infeasible although both of its parents are feasible solutions. The natural way to alleviate the above problem is to make sure that a gene is covered by as few constraints as possible. Formally, the task of optimizing gene layout is to minimize the function $F = \sum_{i=1}^{n} N_{g_i}$ where N_{g_i} is the number of constraints covering the gene g_i.

Example 3.1 *Suppose there are three mutual exclusion constraints:* $E1 = \{1, 4\}$,$E2 = \{2, 3, 5\}$ *and* $E3 = \{1, 6\}$. *It is easy to see from Figure 3(a) that if we just use the layout which places gene* i *at position* i*, there are a lot of overlap among constraints. For example, gene 3 is covered by all three constraints. If a rearrangement of the gene positions is made as shown in Figure 3(b), any gene is covered by one and only one constraint.*

In practice, given the large number of constraints and design variables, the way to find the optimal gene placement also seems to be difficult. Therefore, we have used a single objective GA to a-priori find out a relatively good placement of genes. The multi-objective GA search is then run using this gene layout.

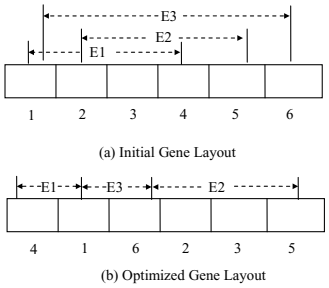


Figure 3. *Optimizing gene layout*

3.2 Handling Invalid Solutions

The simplest way to handle constraint violation is to ignore the solutions that violate any of the constraints. Unfortunately, the number of mutual exclusion constraints is often very high leading to a large fraction of design points being infeasible. In such cases, the naive approach has difficulty in finding even one feasible solution, let alone a set of Pareto-optimal solutions. In order to proceed towards a Pareto front, we need to consider infeasible solutions in some better way rather than ignoring them.

In this paper, we instead *repair* an infeasible solution [11]. The repaired solution replaces the original infeasible solution in the population. There are no standard heuristics for the design of repair algorithms; often people use simple greedy heuristics. We propose the following repair strategy based on branch and bound search: (1) For all the genes involved in violated constraints, clear their values to 0 and (2) Perform a branch and bound search over the genes whose values are cleared to 0 in step 1 to obtain the best feasible solution.

The branch-and-bound repair mechanism retains the "correct" portion of an infeasible solution (the portion of a chromosome which does not violate any design constraints) and replaces the "erroneous" portion with a correct one by performing a search over valid gene value combinations. A repair limit is set to reduce the total number of genes to be repaired. Note that as all the genes involved in violated constraints are cleared to 0, this partial repair always results in feasible solutions.

Within the branch-and-bound search, the search space is pruned by defining heuristic functions to bound the objective values of partially instantiated solutions. Let S denote a partial solution, i.e., all its genes have not been instantiated. If the estimated objective function values for solutions found by extending S are dominated by the objective function values of existing complete solutions, then we do not need to explore any solution obtained by extending the partial solution S.

4 Case Study: Custom Instruction Selection

To illustrate the effectiveness of our hybrid multi-objective GA, we now present a case study with the *custom instructions* selection problem. Selection of custom instructions (computational patterns) [1] constitutes a classic design space exploration problem. Significant research effort has been invested in developing automated selection techniques. However, to the best of our knowledge, all these techniques attempt to optimize for a single objective such as maximizing performance of the application, or minimizing the area required to implement the custom instructions etc. For high-performance, low-cost embedded systems, it is important to optimize for multiple conflicting objectives.

4.1 Problem Formulation

Given an application let us assume, without loss of generality, the following conflicting objectives for the custom instructions selection problem: (1) maximize the performance of the application and (2) minimize the additional area requirement due to the custom instructions. The following problem formulation can of course be easily generalized to include additional objectives such as minimizing the total energy consumed by the application. Given an application, the first step of the design process is to extract all possible computational patterns from the application [14]. Let us assume that we have identified N potential patterns (custom instructions) in a program denoted by $C_1 \ldots C_N$. A pattern C_i has n_i different instances occurring in the program denoted by $c_{i.1}$,...,$c_{i.n_i}$. Let P_i be the performance gain obtained by implementing C_i in hardware as opposed to software. R_i is the amount of area (hardware blocks) required to implement C_i. Let $f_{i.j}$ be the execution frequency of pattern instance $c_{i.j}$ obtained through profiling of the application. Then our goal is to cover each original instruction in the code with zero/one patterns such that the performance is maximized and area is minimized. Formally, for each pattern instance $c_{i.j}$ let us define binary variable $s_{i.j}$ to be equal to 1 if $c_{i.j}$ is selected and 0, otherwise. So the objective functions for the problem are:

$$performance = \sum_{i=1}^{N} \sum_{j=1}^{n_i} (s_{i.j} \times P_i \times f_{i.j}) \quad (1)$$

$$area = \sum_{i=1}^{N} (S_i \times R_i); \quad (2)$$

$$S_i = \begin{cases} 1 & \text{if } \sum_{j=1}^{n_i} s_{i.j} \geq 1 \\ 0 & \text{otherwise;} \end{cases}$$

[1] We use the terms patterns and custom instructions interchangeably.

In other words, binary variable S_i $(1 \leq i \leq N)$ is equal to 1 if any of the instances corresponding to pattern C_i is selected and 0 otherwise.

The constraint for this problem is that a static program instruction can be covered by at most one custom instruction instance. So, if instances $c_{i_1.j_1}$,..., $c_{i_k.j_k}$ cover a static program instruction, then the corresponding constraint is

$$s_{i_1,j_1} + \ldots + s_{i_k,j_k} \leq 1 \quad (3)$$

This is the *mutual exclusion* constraint which creates difficulty in multi-objective design space exploration. It is easy to see that the single-objective version of this optimization problem (e.g., maximizing performance defined by Equation 1 under a *fixed* area budget) can be easily formulated using Integer Linear Programming or ILP [13]. However, ILP is not practical for large programs due to the excessively long execution time and it cannot be used to optimize against multiple objectives.

4.2 A Multi-Objective Genetic Algorithm

Algorithm 1: Multi-Objective GA with Repair

```
1 geneLayout();
2 initRandomPopulation(); makeFronts(population);
3 for 1 to MAX generations do
4     proportionateSelection(); crossover(); mutation();
5     for each infeasible solution S do
6         branch-and-bound repair(S);
7     makeFronts(newPopulation); rudolphElitism();
```

Algorithm 1 shows our hybrid multi-objective GA algorithm for the custom instruction selection problem. First, we use a single objective GA to find out the best possible gene layout to minimize the average number of constraints covering a gene as we have discussed in Section 3.1. The $makeFronts$ function sorts the population into various fronts on the basis of their non-dominance. After sorting into equivalence sets or fronts, we assign dummy fitness values to the chromosomes according to the NSGA (Non-dominated Sorting Genetic Algorithm) Scheme [5]. The dummy fitness values are assigned in such a manner that all the individuals in front i get higher fitness values than all the individuals in front $i + 1$. Fitness values are used for a proportionate selection of chromosomes. Single-point crossover and a random gene flip mutation are used as basic evolutionary operators [5]. To preserve elitism in successive populations, we use the $rudolphElitism$ function, which picks the best non-dominated solutions from both the parent and off-spring populations [12].

In repairing an infeasible solution, we employ multi-objective branch and bound search. The branch and bound

algorithm instantiates the cleared genes one by one. However, to avoid exploring the entire design space corresponding to these cleared genes, a heuristic function is required. The heuristic function computes an upper bound of performance gain and lower bound of area corresponding to a partially instantiated chromosome. If according to the heuristic function, a partially instantiated chromosome x is dominated by some other fully instantiated chromosome, then we do not need to instantiate x any further. Clearly, the amount of pruning critically depends on the design of the heuristic function. The two heuristic functions used are:

$$performanceGain = \sum_{i=1}^{N}\sum_{j=1}^{n_i}(s'_{i,j} \times P_i \times F_{i,j}) \quad (4)$$

$$area = \sum_{1}^{n}(S'_i \times R_i); \quad (5)$$

$$s'_{i,j} = \begin{cases} s_{i,j} & \text{if } c_{i,j} \text{ is instantiated;} \\ 1 & \text{if } c_{i,j} \text{ is not instantiated and } s_{i,j} = 1 \\ & \text{does not conflict with instantiated } c_{i,j}\text{s;} \\ 0 & \text{otherwise;} \end{cases}$$

$$S'_i = \begin{cases} 1 & \exists 1 \leq j \leq n_i \; s_{i,j} = 1 \text{ and } c_{i,j} \text{ is instantiated} \\ 0 & \text{otherwise;} \end{cases}$$

Intuitively, the heuristic function $performanceGain$ gives an upper bound on performance that a partially instantiated solution can take and $area$ gives an lower bound on the area.

5 Experimental Evaluation

In this section, we evaluate our hybrid multi-objective GA algorithm with custom instruction selection problem. We use six benchmark programs selected from MediaBench and MiBench. We use SimpleScalar tool set for the experiments. The programs are compiled using gcc 2.7.2.3 targeted for SimpleScalar with -O3 optimization. The GA algorithms are run on a Sunfire 4800 server containing eight 750MHz Ultra Sparc III CPU with 8 GB RAM.

Given a binary executable of an application, we first exhaustively enumerate all possible patterns and their instances [14]. We impose a constraint of maximum 4 input operands and 2 output operands for any pattern. The execution frequencies of the pattern instances are obtained through profiling. The hardware latencies and area of custom instructions (patterns) are obtained using Synopsys synthesis tool. Finally, the number of execution cycles of a custom instruction is computed by normalizing its latency (rounded up to an integer) against that of a multiply-accumulate (MAC) operation, which we assume takes exactly one cycle. We do not include floating-point operations, memory accesses, and branches in custom instructions as they introduce non-deterministic behavior.

Max. Area (gates)	Area		Speedup (%)	
	Heuristic	GA	Heuristic	GA
2,000	1,971	1,772	4.7	4.8
4,000	3,992	3,992	6.1	8.3
6,000	5,650	5,963	7.1	10.2
8,000	7,432	7,556	7.8	12.1
10,000	9,631	9,338	9.1	13.4
12,000	11,795	11,411	9.9	14.0
14,000	13,811	13,926	10.7	14.9
16,000	15,884	15,499	11.7	15.9
18,000	17,695	17,100	12.3	16.4

Table 1. *Comparison of GALayoutRep with a heuristic method for djpeg benchmark.*

Effectiveness of layout and repair To evaluate the effectiveness of gene layout and repair, we compare the performance of the following three genetic algorithms.

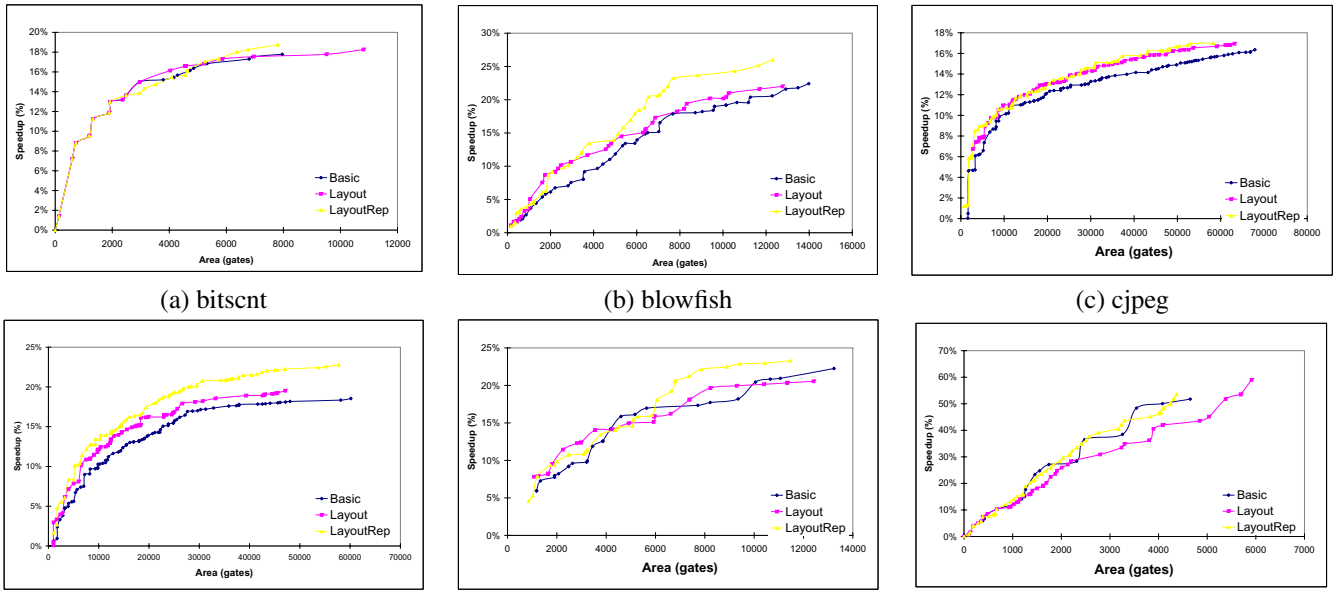
- *BasicGA*: GA which uses a random gene layout and does not repair infeasible solutions.
- *GALayout*: GA which optimizes gene layout but does not repair infeasible solutions.
- *GALayoutRep*: GA which optimizes gene layout and also repairs the infeasible solutions.

For each of the above algorithms, we run for 200 generations with population size 100. We choose a cross-over probability of 0.5 and mutation probability of one gene per solution. In branch and bound repair process, we repair at most 20 genes in an infeasible chromosome.

The Pareto front generated by the three algorithms are shown visually in Figure 4. The objective function *Speedup* appears along the y-axis and *area* appears along the x-axis (the unit of *area* is the number of gates). The figure clearly shows the superiority of *GALayoutRep* that optimizes the layout and repairs infeasible solutions.

The time required to optimize the gene layout is negligible (at most 0.23% of the time required by *BasicGA*). To investigate the time overhead of *GALayoutRep*, we ensure that *GALayoutRep* runs for the same amount of time as *Basic GA*. The number of generations used by *GALayoutRep* for various benchmarks now varies from 160 to 170 to take into account the time taken by repair. Even with this restriction, we observe that *GALayoutRep* is still superior.

Comparison of GA with heuristic methods Finally, we compare the solution returned by multi-objective GA with state-of-the-art heuristic methods. As mentioned before, previously proposed solutions are all based on single objective. To convert multi-objective version of the optimization problem into a single-objective version, we seek to optimize performance under a fixed area constraint. For GA, we still



(a) bitscnt (b) blowfish (c) cjpeg

(d) djpeg (e) rijndael (f) sha

Figure 4. *Visual comparison of Pareto fronts*

use multi-objective *GALayoutRep* to create the Pareto front; we then select the solution from the Pareto front that leads to highest speedup under the area constraint. We compare it against the speedup of the solution returned by heuristic method proposed in [13]. The heuristic method ranks the pattern instances according to $speedup/area$ ratio. It selects patterns according to this rank ensuring that the mutual exclusion constraints are not violated. Table 1 shows the comparison of GA with heuristic for *djpeg* benchmark. Other benchmarks show similar trends. GA achieves higher speedup than the heuristic solution; in some cases it performs better in both area and performance objectives.

6 Discussion

In this paper, we propose a hybrid multi-objective search algorithm for design space exploration, namely multi-objective GA combined with multi-objective branch and bound for repair. Apart from pruning sub-optimal parts of the design space, we are concerned about GA getting stuck with a large number of infeasible solutions as the search progresses. Our repair strategy takes care of this concern. We employ our algorithm to the design of application-specific instruction set extensions. Experimental results from this design problem show that the quality of the Pareto front improves substantially due to our repair strategy.

Acknowledgments This work was partially supported by a InfoComm and InfoTech Initiative (ICITI) project R252-000-150-112 at the National University of Singapore.

References

[1] K. Apt. *Constraint Programming*. Cambridge University Press, 2003.

[2] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *DAC*, 2003.

[3] N. Clark et al. Processor acceleration through automated instruction set customization. In *MICRO*, 2003.

[4] D. Dasgupta and Z. Michalewicz. *Evolutionary Algorithms in Engineering Applications*. Springer Verlag, 1997.

[5] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley, 2001.

[6] M. Eisenring et al. Conflicting criteria in embedded system design. *IEEE Design and Test*, 17(2), 2000.

[7] W. Fornaciari et al. A sensitivity-based design space exploration methodology for embedded systems. *Design Automation for Embedded Systems*, 7(1-2), 2002.

[8] T. Givargis and F. Vahid. Platune: A tuning framework for system-on-a-chip platforms. *IEEE TCAD*, 21(11), 2002.

[9] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2), 2000.

[10] V. Kathail. PICO: Automatically designing custom computers. *IEEE Computer*, 35(9), 2002.

[11] Z. Michalewicz. A survey of constraint handling techniques in evolutionary computation methods. In *Annual Conf. on Evolutionary Programming*. MIT Press, 1995.

[12] G. Rudolph. Evolutionary search under partially ordered sets. Technical Report C1-67/99, Univ. of Dortmund, 1999.

[13] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *DAC*, 2004.

[14] P. Yu and T. Mitra. Scalable custom instruction identification for instruction-set extensible processors. In *CASES*, 2004.