# A Framework for User Assisted Design Space Exploration

X. Hu[a] G. W. Greenwood[b] S. Ravichandran[b] G. Quan[a]

[a] *Dept. of Computer Science & Engineering, University of Notre Dame, Notre Dame, IN 46556*
[b] *Dept. of Electrical & Computer Engineering, Western Michigan University, Kalamazoo, MI 49008*

**Submit to** *DAC'99*

**Contact author:** Dr. Xiaobo (Sharon) Hu

Phone: (219) 631-6015
Fax: (219) 631-9260
E-mail: shu@cse.nd.edu

**Presenter:** Xiaobo (Sharon) Hu
**Topics:** M1.6, T5.2

**Statement:** All appropriate organizational approvals for the publication of this paper have been obtained. If accepted, the authors will prepare the final manuscript in time for inclusion in the Conference proceedings and will present the paper at the conference.

**Signed by** X. Hu, G. W. Greenwood, S. Ravichandran, G. Quan

# A Framework for User Assisted Design Space Exploration

**Abstract**

Much effort in hardware/software co-design has been devoted to developing "push-button" types of tools for automatic hardware/software partitioning. However, given the highly complex nature of embedded system design, user guided design exploration can be more effective. In this paper, we propose a framework for designer assisted partitioning that can be used in conjunction with any given search strategy. A key component of this framework is the visualization of the design space, without enumerating all possible design configurations. Furthermore, this design space representation provides a straightforward way for a designer to identify promising partitions and hence guide the subsequent exploration process. Experiments have shown the effectiveness of this approach.

## 1 Introduction

Today's competitive marketplace demands that increasingly complex computer systems be developed in increasingly shorter time periods. There are a number of factors such as ambiguous specifications, product bugs, non-availability of parts, and so on which all contribute to a suboptimal design environment. Yet, even if one were able to virtually eliminate these factors, there is still the issue of design complexity. Many consumer products are now being produced as embedded systems-on-a-chip. These systems require hardware and software to be designed in ways which brings about easier hardware-software integration. In many cases, manual procedures have only a limited ability to effectively address this issue. Co-design, however, does show promise in this area.

Ideally, co-design should allow designers to make decisions early in the development cycle regarding how tasks should be partitioned between hardware and software components [1, 2]. Equally important is the ability to rapidly determine the performance resulting from a particular partition. Successful co-design requires that specification, partitioning, and co-verification all be appropriately addressed. In this paper we will deal only with the partitioning issue—one of the challenging aspects of co-design. It is important to emphasize the importance of correct partitioning. Designers rapidly move to the implementation phase once a partition has been defined. Improper partitioning leads to constrained designs which are difficult (and costly) to change in the future.

A number of interesting research results have been published, which attempt to produce tools or methodologies that automatically partition tasks among processors and other hardware components (e.g., [3, 4, 5, 6, 7, 8, 9, 10]. Although these papers describe different levels of abstraction or different system organization, and use different optimization algorithms, the underlying goal is to provide a "push button" design environment. Several electronic design automation vendors are currently attempting to develop automated partitioning tools [11]. Although it is desirable to have these tools take over menial, low-level tasks associated with partitioning, it can be much more effective for designers to retain certain control of the complex design process. Incorporation of a designers expertise, done in an intelligent manner, will improve the partitioning process in terms of both speed and quality.

The space of all possible hardware-software partitions is enormous, which requires an intelligent search methodology—something most effectively provided by a designer. This paper describes a framework

for user assisted partitioning that can be used in conjunction with any given search strategy. One key component of this framework is a visual means of depicting the design space, without enumerating all possible design configurations. Furthermore, this design space representation provides a straightforward way for a designer to identify promising partitions without worrying about the underlying, low-level details of the design. It will also be shown how user feedback helps to identify partitions that produce robust designs.

The paper is organized as follows. Section 2 describes the problem domain and specifically illustrates the complexity of searching for good partitions. Section 3 gives a detailed discussion of our approach for capturing and depicting certain characteristics of the solution space that are essential for user guided search. Section 4 describes our proposed framework and how it is used to provide feedback, which guides the search for good partitions. Section 5 shows how the framework is used on a real-world example taken from the automotive industry. Finally, Section 6 summarizes our work and discuss future extensions.

# 2 Preliminaries

This section provides essential definitions of the problem of interest, fitness landscapes, and the complexity of searching for solutions to combinatorial optimization problems.

## 2.1 Problem Domain

The approach to hardware/software partitioning that we take is at the system level. It is our belief that system level partitioning should address some of the following questions:

1. Which functions should be implemented in dedicated hardware circuits and which should be in software?
2. What processors and ASICs should be used, and which software tasks should be executed by which processor?
3. Can an identified system configuration meet all of the temporal requirements, as well as other constraints such as component compatibility?

Such a high level of abstraction allows exploration of hardware architectures as well as hardware/software partitions [8, 10, 12]. Our experience indicates that these combined decisions are the primary factors that define the cost and performance of an embedded system [13].

In system-level design, system specifications are modeled as a collection of functions, $\mathcal{F}$, (also referred to as tasks in some literature). Associated with each function, there are generic measurement of the complexity of the function (such as instruction counts and memory requirements), timing constraints (such as a deadline and a period), and other performance related requirements. The set of requirements and constraints are stated in the system specifications.

Each function may be implemented in either hardware or software. The candidate hardware components are collected in a hardware library $\mathcal{H}$ as a list of hardware modules. For each hardware module, the types of functions it implements are specified. For example, an ASIC module may implement one

or more control functions from the specification, and a microprocessor module implements a generic CPU function which indicates that it can execute any software modules assigned to it. Performance data such as execution speed, power consumption and cost can be specified for each hardware module. Note that the hardware library can include not only existing components but also hypothetical ones for those not yet developed.

For those functions that may be implemented in software, respective software modules are constructed and collected in a software library $\mathcal{S}$. Each module includes the information on the functions it implements and its complexity measured by execution time or instruction counts when executed on a particular processor. If a function requires different execution times when running on different processors, a software module for the function can be constructed for each processor, so that processor-dependent performance is accounted for. Although estimating the processing time of software modules on a wide variety of different implementations of different architectures is not practical, we believe these estimates are possible as long as the choice of architectures is limited. By establishing standard benchmarks that characterize the performance of a processor for different instruction mixes, and by analyzing the instruction mix of software modules, we have found it possible to reasonably predict the performance of software modules over different implementations of several different processor families.

To find an *optimal* implementation for a given system specification, we need to quantify *optimality*. As we pointed out previously, several attributes are used to gauge the quality of a system, e.g., cost, power consumption and system expandability. A straightforward approach is to use a weighted sum to model the tradeoffs among the various attributes. Consequently, the optimization problem becomes one of finding an implementation $\mathcal{X}_{opt}$ for the specified functions $\mathcal{F}$ such that the hardware and software components are modules from the given libraries $\mathcal{H}$ and $\mathcal{S}$ and the composite attribute value is optimized. Of course, the constraints associated with each function should be met by each feasible implementation.

It is important to realize that the above issues are equally important in existing designs that must be modified. These modifications may result from customer requests, correction of bugs, or normally scheduled upgrades.

### 2.2 Fitness Landscapes and Search Complexity

The partitioning problem described in the previous section is an example of a combinatorial optimization problem. There has been considerable recent interest in the design of efficient algorithms which are capable of finding good solutions to these types of problems. Implicit is the idea that the solutions to optimization problems reside in an abstract solution space and two solutions are neighbors if they differ by a single mutation of a problem parameter. Associated with each solution is a real number that reflects fitness or quality of that solution. This space and the associated fitness values form a fitness landscape[1]. Any algorithm that "solves" an optimization problem is therefore a search algorithm that explores the fitness landscape. Put another way, optimization problems are search problems.

The $NK$ family of rugged, multipeaked fitness landscapes were introduced to help examine the difficulties in exploring fitness landscapes in general [14]. The model consists of $N$ problem parameters

[1] In practice, fitness will be with respect to one or more attributes such as cost or power consumption; high fitness is associated with good values of the attribute.

which define a solution, and $K$ corresponds to the number of other problem parameters related. In other words, each problem parameter contributes to the overall fitness of the solution, but each parameter does so by interacting with exactly $K$ other parameters within the same solution. If $K = 0$, there is no parameter interaction and the landscape is smooth with few local optima. As $K$ approaches its maximum value of $N - 1$, the landscape becomes increasingly rugged with a large number of local optima.

This $NK$ model has direct application to the partitioning problem. The problem parameters are the tasks and how they are allocated among the given set of hardware components. The effect of one parameter's allocation on the overall fitness (say cost) is dependent on whether this task must share resources with other tasks. Therefore, $K$ for the partitioning problem can be quite large, and the corresponding fitness landscape can be both highly dimensional and quite rugged. This precludes an exhaustive search for optimally fit solutions. Indeed, the system design problem is NP-complete [15] which means finding the optimal solution in finite time is unlikely. This has lead researchers to adopt heuristic search strategies.

One obvious method of reducing the level of effort required to search the fitness landscape is to reduce the size of that landscape by removing some of the problem parameters (which reduces the dimensionality) or by reducing the number of values a problem parameter may assume. This smaller search space, in principle, should be easier to search. Some caution is warranted—the reduced search space may still contain an exponential number of points!

A more effective approach is to construct an "intelligent" search operator that is constrained to search in only specified regions on the fitness landscape. All heuristic search operators take current solutions and perturb the problem parameters in some stochastic manner to produce new solutions for evaluation. This intelligence can be manifested by *i*) constraining search operators to make only specific modifications to specific problem parameters, or *ii*) penalizing solutions from undesirable regions by artificially reducing their natural fitness values [16]. Either technique forces the search algorithm to explore in a reduced solution space.

The most effective way of constructing an intelligent search operator is to actively involve the designer in the search process. The designer assists the search process by identifying regions on the fitness landscape to avoid or regions to explore more thoroughly. The user must be able to quickly identify critical regions in the landscape, and the feedback to the search process must be done in a clean, straightforward manner. The best way of presenting information to the user is via a graphical image of the landscape—something not easy to do since the landscape is normally a high-dimensional, multimodal entity with an exponential number of points! Nevertheless, we have developed a technique which constructs a 3D image of the fitness landscape regardless of how many dimensions actually exist. In the following sections, we present our technique for depicting the topology of the fitness landscape. It will then be shown how this knowledge can be exploited to guide the search process.

## 3 Landscape Representation

This section discusses our technique for constructing a 3-D landscape from a high-dimensional landscape.

### 3.1 Portraying Fitness Landscapes in 3 Dimensions

Every adjustable design parameter in a partitioning problem adds another degree of freedom to the problem. Examples include the choice of processors, the number of ASICs, hardware/software partitions, and so on. Each degree of freedom adds increasing levels of complexity and each adds another dimension to the fitness landscape. Real-world instances of partitioning problems can be expected to have far more than 3 dimensions making the fitness landscape difficult to visualize. One-dimensional techniques such as computing correlations from random walks have been presented as a means of characterizing these high-dimensional landscapes [17]. Unfortunately, correlation computed from a single random walk is error prone because the landscape is likely to be anisotropic [18]. We have developed a mapping technique that can graph higher dimensional landscapes. Admittedly our approach also has a shortcoming typical of 1-dimensional techniques—information contained in higher dimensions is lost. Nevertheless, it does present more information than does a simple correlation, and it does graphically reflect landscape ruggedness. The basic concept relies on the ability to isomorphically embed a lattice into a $m$-ary $n$-cube. First, however, it is necessary to digress for some definitions.

Let $b_1 b_2 \ldots b_n$ be a $n$-bit binary string where $b_i \in \{0, 1\}$. There are $2^n$ unique binary patterns that can be formed with $n$-bit binary strings. A sequence of length $L$ contains $L$ $n$-bit binary strings no two of which are identical. This sequence is a *Gray code sequence* if any two successive strings differ in one and only one bit position. Note that this requirement must also hold between the first and last $n$-bit strings in the sequence. For example, $\{00, 01, 11, 10\}$ and $\{00, 01, 10, 11\}$ are both sequences of 2-bit binary strings but only the first one is a Gray code sequence.

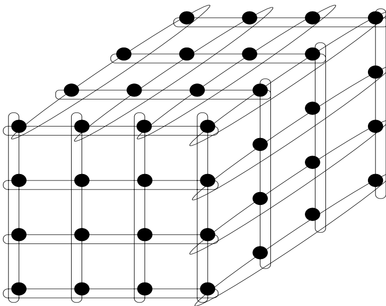

Figure 1: A $k$-ary $n$-cube with $k = 4$ and $n = 3$. Hidden nodes and edges are not shown to preserve clarity.

Without loss in generality, we assume in the $m$-ary $n$-cube that $m$ is an integer power of two. Each node in the $m$-ary $n$-cube is labeled with a $n \lg m$ bit binary label. The labeling is done in a Gray code manner such that any two nodes connected by an edge differ in only one bit position. In other words, the labels identify 1-mutant neighbors.

Let $A$ and $B$ be integer powers of two. The binary label associated with each node in the $m$-ary

$n$-cube can be partitioned into two parts which is written in the form

$$\{\underbrace{b_1 \dots b_r}_{\lg A}\ \underbrace{b_{r+1} \dots b_{n \lg m}}_{\lg B}\}$$

Thus any point $(x, y)$ in the 2-dimensional lattice is given by using the $\lg A$ most significant bits to define the $x$ coordinate, and the $\lg B$ least significant bits to define the $y$ coordinate. Since the embedding of the lattice is isomorphic in the $m$-ary $n$-cube [19], any two adjacent nodes in the cube are also adjacent in the lattice. Figure 2 shows a portion of an $4 \times 16$ lattice embedding into the 4-ary 3-cube of Figure 1.

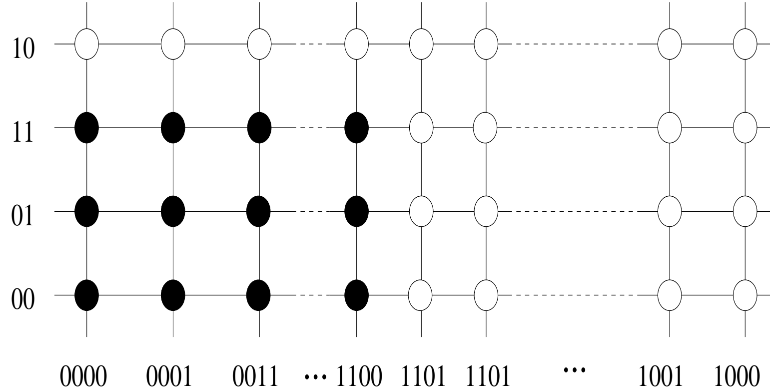


Figure 2: A $4 \times 16$ grid embedding into a 4-ary 3-cube. The rows are assigned the $\lg A$ most significant bits and the columns are labeled with the remaining bits of the binary label.

In effect, this embedding has "unfolded" a high dimensional cube into a 2-dimensional lattice. This unfolding process does break edges in the cube and so some neighbor relationships are lost. Nevertheless, a fraction of the neighbor relationships are preserved; adding fitness now forms a 3-dimensional graph of the fitness landscape. A step-by-step algorithm for constructing this 3-dimensional graph is available in [20]. In Section 5 we will show several of these fitness landscapes. But first, we need to discuss neighborhood relationships for partitioning problems.

### 3.2 Who Are My Neighbors?

The data structure for a partitioning problem can be represented by a binary string where distinct fields represent distinct design parameters. For example, in a problem with $N$ tasks, a design solution could be represented by a $k \cdot N$ bit string, where each task has $k$ bits to define one of $2^k$ possible hardware modules that it could be assigned to for execution. If the $k$-bit encoding were done in a Gray code manner, then two hardware modules that differ in only one bit position would be considered 1-mutant neighbors.

As discussed in the previous section, each design parameter adds a dimension to the fitness landscape. Since there are $n$ total parameters with up to $k$ total values per parameter, the original fitness landscape is a homomorphism on a $k$-ary $n$-cube. This high-dimensional landscape is then mapped into two dimensions and a third dimension, which represents fitness, is added. For the moment, consider just

one design parameter. With a little thought it should become apparent that the juxtaposition of values for the design parameter defines the distribution of fitness values, which also defines the topology of the fitness landscape.

There are no universal rules for defining 1-mutant neighbors that will apply in all situations. In principle, smooth fitness landscapes should be easier to explore than rugged landscapes. There are some general guidelines which we have found will tend to keep the fitness values of 1-mutant neighbors from becoming drastically different. Processors often come in several different versions, but the only significant difference is in the clock speed or amount of on-chip memory. Try to keep processors with similar clock speeds as neighbors and then order them according to the amount of on-chip memory. ASICs with a similar number of programmable cells and similar pin-outs should be neighbors. In other words, modules that have similar attributes should be placed as 1-mutant neighbors.

Another way of defining neighbors can ultimately lead to the identification of robust designs. By robustness, we mean that a design is more tolerant to perturbations in the modules used. In this case, 1-mutant neighbors are those which involve minimal redesign effort. Robust designs will therefore reside in regions of relative smoothness in the fitness landscape. For example, suppose a processor is no longer available and must be replaced. It will generally require virtually no redesign effort to replace this processor with one that has an identical core, and perhaps a larger amount of on-chip memory. Moreover, this new processor should have similar values for its attributes (e.g., power), which should result in a similar fitness in the designs. By placing these processors as neighbors on the fitness landscape, it should be easy to verify robustness by examining the smoothness in the landscape.

## 4 Landscape Guided Search

This section provides detailed information about our user-assisted framework for exploring design solution space. Figure 3 shows the framework of our proposed design environment. The user plays a role in three key areas:

1. *construction of the solution space*

   The solution space can be constructed once a data structure and fitness function have been defined. This information is derived from the system specification. The hardware and software libraries provide the design parameters for the data structure. The user is responsible for "ordering" this library data to establish the neighborhood relationships. The user also defines the fitness function using information extracted from the system specifications.

2. *construction of the fitness landscape*

   The fitness landscape is normally constructed with respect to only one attribute (e.g., power). Complete enumeration of the fitness landscape is rarely feasible due to the large number of potential solutions. Therefore, a "quantized" landscape is often depicted (see below). The user specifies the degree of quantization and can zoom in on specified regions for greater detail.

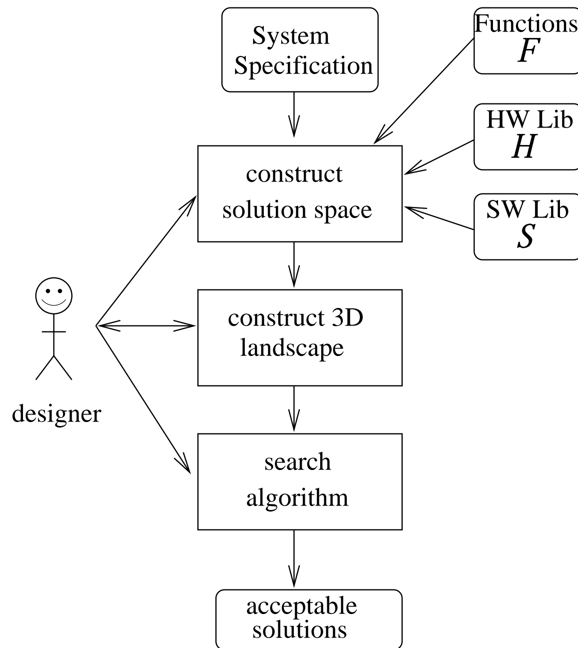


Figure 3: Overview of the design framework. User provides guidelines for constructing neighbors to assist construction of solution space and the scale factor for the 3D landscape depiction. Information obtained from the scaled landscape allows the user to develop guidelines for constraining the search algorithms to specified regions of the solution space.

3. *guiding the search process*

   Figure 4 shows a representative 3-dimensional landscape. (This particular landscape is with respect to the attribute of power. See Section 5 for further details.) Notice that the X-Y axis are expressed as integers—a technique that hides the underlying details of the solutions. This approach was intentional. The user need only identify regions on the X-Y plane that warrant a detailed exploration. Conversely, the user can specify regions to avoid. This information is provided to the search algorithm, which automatically designs search operators that are constrained to search (or avoid) regions identified by the user.

As stated earlier, complete enumeration of the fitness landscape of a partitioning problem is time consuming and often computationally impossible. However, if the solution space is formed following the guidelines given in Section 3, such complete enumeration is not necessary. A fitness landscape can be depicted with varying levels of granularity. The user can specify a quantization degree which plots only a subset of the points. For example, the user may only want every tenth point on each axis plotted. The quantized landscape illustrates only the prominent landscape features. Nevertheless, often this will be a sufficient level of detail to identify regions of low fitness, which should be avoided during the search process. Furthermore, if the user likes to investigate some regions in greater detail, s/he can use the X-Y coordinates to “zoom in” on the selected regions.

Note that each solution in the fitness landscape is assigned unique X-Y coordinates. Using integers on the axes provides a natural method for a user to select regions for further exploration or avoidance without worrying about specific parameter values—only intervals on the X and Y axis need to be provided to the search algorithm. The search process itself can take place in the X-Y plane, hiding the underlying details from the user. Monte Carlo techniques and evolutionary algorithms are ideally suited for searching this type of fitness landscape. A mapping from the X-Y plane to the corresponding solution space must be done in order to determine the fitness value. Nevertheless, this mapping can be done in a straightforward manner.

To illustrate the search process, suppose a Monte Carlo search algorithm is being used. The current solution would have X-Y coordinates of, say, (62, 34). A new solution could be generated by stochastically perturbing X and/or Y. If the new X-Y values are in a region to avoid, the solution should be discarded and a new candidate solution created. The X-Y values should then be converted into binary patterns (see Section 3.2) and subsequently decoded to calculate the actual fitness. If the candidate solution has a lower power consumption—which denotes higher fitness—it replaces the existing solution. Otherwise, it is accepted with an exponentially decreasing probability.

A couple of items are particularly worth noting. First, restricting the search algorithm to explore X-Y space hides the details of the underlying solutions beings evaluated. More specifically, the search algorithm is only concerned with exploring an integer space which is completely independent of the complexity of the design task. Secondly, the user is not burdened with identifying design restrictions via cumbersome if-then-else rules or some other type of abstraction; design constraints are provided by just listing avoidance regions in terms of X-Y coordinates—a process which is both natural and universally applicable.

## 5 An Example

An example will help to fix ideas. Consider a design problem with 9 total tasks and 16 hardware modules which includes both processors and ASICs. This example is taken from an engine control module design described in[12]. Information about the 9 tasks and the hardware modules is found in Table 1 and Table 2, respectively. (More explicit description of this example can be found in [12].) The aim is to determine whether each task should be implemented in hardware or software and which modules should be included in the final system.

For this example, the search itself was conducted by a software package we had previously developed called **EvoC**, which uses an evolutionary algorithm for exploration. Details on **EvoC** can be found elsewhere [12]. The focus here is not on the specifics of a particular search algorithm, but rather on how our 3D fitness landscape representation assists the designer in guiding the search for good solutions. It is important to re-emphasize that our landscape representation is not restricted to only certain types of search algorithms. Indeed, it is entirely independent of the search mechanism.

In one experiment for this example, we considered a fitness landscape based upon power consumption alone. (Note that our approach can combine more than one landscape to limit the search space. For simplicity, we only discuss the single attribute case here.) Figure 4 depicts the fitness landscape using

| Name | Deadline | Period | Activation | Instructions |
|---|---|---|---|---|
| DigitalFilter1 (DF1) | 46.00 | 104.17 | 0.00 | 64 |
| DigitalFilter2 (DF2) | 10000.00 | 10000.00 | 9895.83 | 32 |
| DecodeSPUB (DSB) | 83.00 | 208.33 | 0.00 | 30 |
| DecodeSPUA (DSA) | 138.00 | 208.33 | 83.00 | 30 |
| ReadCAM (RC) | 416.67 | 10000.00 | 0.00 | 30 |
| ServiceRoutine (SR) | 208.33 | 416.67 | 0.00 | 20 |
| FuelCalc (FC) | 1333.33 | 2500.00 | 833.33 | 480 |
| SparkCalc (SC) | 2500.00 | 2500.00 | 1666.67 | 100 |
| ReadMAP (RM) | 312.50 | 416.67 | 0.00 | 40 |

Table 1: Specification of a subset of engine control tasks. Deadline, period, and activation are all in units of $\mu s$. The number of instructions are based on a generic instruction set.

a scale factor of 3000 for both X and Y coordinates. Since lower power consumption implies higher fitness, fitness is proportional to the inverse of power consumption. Notice there are numerous regions that contain sinkholes, which are regions where solutions have a power consumption exceeding 20W. All solutions residing in sinkholes have their fitness value fixed to a constant of 0, regardless of the actual value. This value is deemed low enough so that a search algorithm would discard the solution. In practice, the user could tell the search algorithm to avoid certain regions in the fitness landscape by meerly inputting X-Y coordinates (e.g., "avoid for all $X$, $0.5 \times 10^5 < Y < 1.5 \times 10^5$").

A stochastic search operation should be more computationally efficient if the search algorithm knows *a priori* which regions to avoid. To test this hypothesis and hence illustrate the value of user guided design exploration, we conducted a series of search operations using **EvoC** in which avoidance regions were identified in some runs, but not in others. To make the comparisons fair, all runs started with the same initial conditions and results were averaged over 20 runs. One set of search results from **EvoC** are depicted in Figure 5, where the goal is to minimize power consumption. The figure shows that the convergence rate has improved by specifically avoiding regions of known low fitness. Given that only a single avoidance region is used in this example— and the overhead for doing this quite low—the improvement is quite appreciable. Another set of search results are shown in Figure 6, where the goal is to minimize a weighted sum of scaled cost and power consumption (with weights of 0.4 and 0.3, respectively). We scaled the cost and power in such a way that the fitness values are within 0 and 1 and that the higher the fitness, the better the solution. Again, an improvement in the convergence rate can be seen from Figure 6,

In another experiment for the engine control example, we investigated a fitness landscape based upon feasibility factor alone. The feasibility factor of a system measures the capability of the system

| **Name** | **Core/Tasks directly Implemented** | **Cost** | **Power Consumption** | **MIPS Available** |
|---|---|---|---|---|
| MC1-H | CPU, DF1,DF2,DSB,DSA | 3.50 | 5.00 | 1.30 |
| MC2-H | CPU, TC(32) | 3.25 | 14.00 | 1.50 |
| MC3a-H | CPU, TC(16) | 5.25 | 1.30 | 2.50 |
| MC3b-H | CPU, DF1,DF2, DSB,DSA | 6.25 | 1.00 | 2.50 |
| MC4a-H | CPU, DF1, DF2, DSB, DSA, TC(14) | 3.75 | 0.30 | 1.70 |
| MC4b-H | CPU, DF1, DF2, DSB, DSA, TC(14) | 3.25 | 1.50 | 1.35 |
| MC4c-H | CPU, TC(16) | 2.50 | 1.50 | 1.70 |
| P1-H | CPU, | 2.00 | 14.00 | 1.43 |
| P2-H | CPU | 13.00 | 13 | 13.50 |
| P3-H | CPU, | 3.50 | 10.00 | 9.30 |
| P4-H | CPU, | 4.25 | 5.00 | 2.50 |
| ASIC1-H | DF1,DF2,DSB,DSA | 2.50 | 1.65 | - |
| ASIC2-H | DF1,DF2,DSB,DSA | 3.00 | 1.20 | - |
| ASIC3-H | DF1,DF2,DSB,DSA | 2.85 | 1.80 | - |
| PIO1-H | TC(16) | 1.00 | 1.00 | - |
| PIO2-H | TC(16) | 0.80 | 1.30 | - |

Table 2: Specification of the hardware modules to implement the tasks in Table 1. TC is a timing channel and MIPS is a measure of the processing power of a CPU in millions of instructions per sectiond.

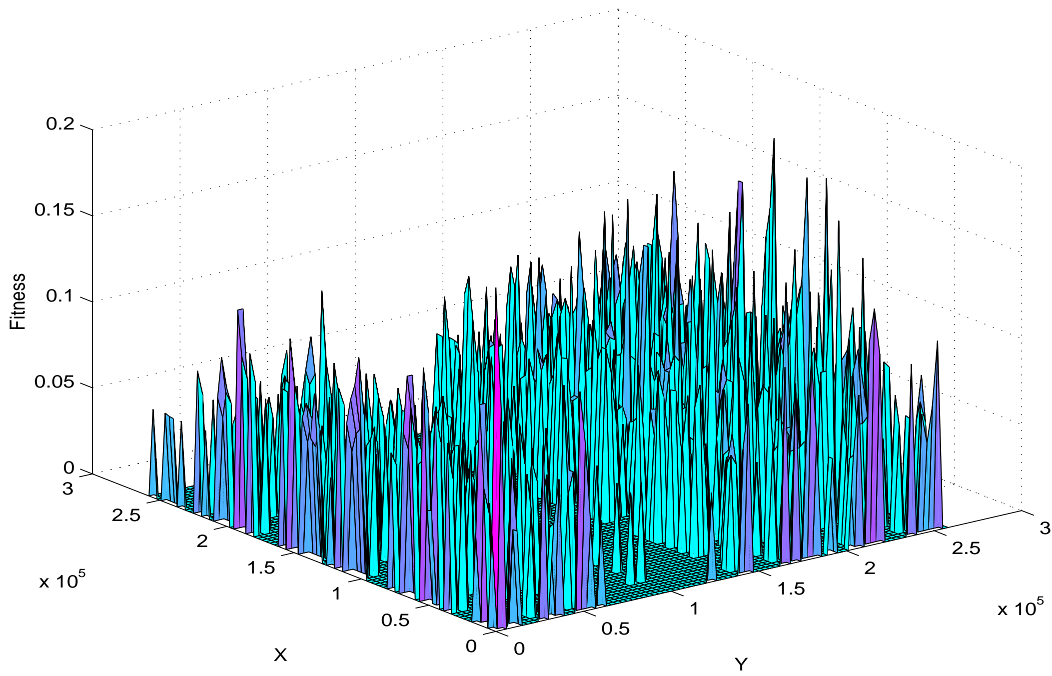


Figure 4: A fitness landscape for the power consumption attribute.

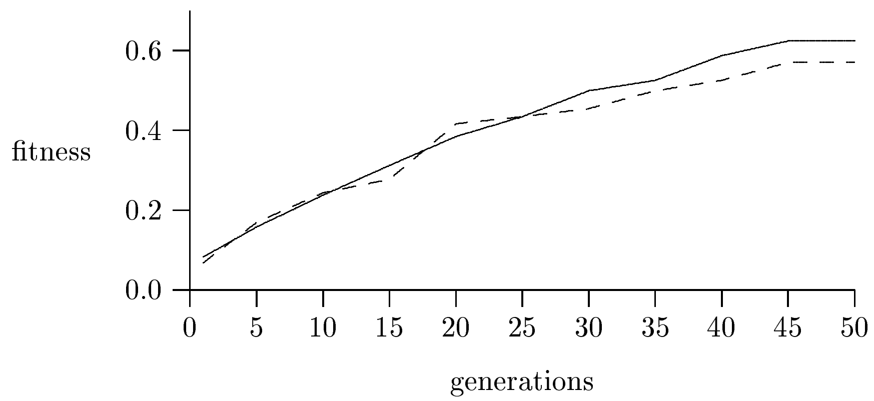


Figure 5: Fitness versus number of generations. Depicted results are averaged over 10 runs of **EvoC**, searching on the power consumption fitness landscape representation. The solid (dashed) line is for searches conducted with (without) constraints. The fitness value scaling is arbitrary.

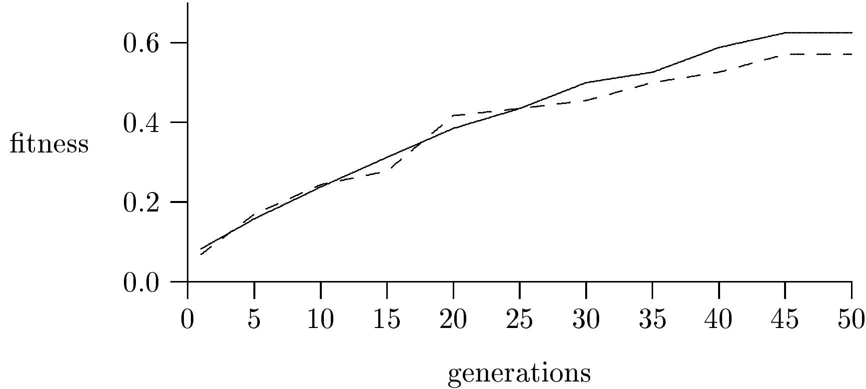


Figure 6: Fitness versus number of generations. Depicted results are averaged over 10 runs of **EvoC**, searching on the power consumption fitness landscape representation. The solid (dashed) line is for searches conducted with (without) constraints.

to satisfy the real-time requirements [21], and higher feasibility value implies high fitness. Figure 7 depicts the fitness landscape using a scale factor of 3000 for both X and Y coordinates. Notice there are numerous regions that contain sinkholes, which are regions where solutions have a feasibility factor value less than 0.7. Again, all solutions residing in sinkholes have their fitness value fixed to a constant of 0, regardless of the actual value. Based on the landscape information, we conducted a search which avoids the region of $1.3 \times 10^5 < X < 2.3 \times 10^5$ for any $Y$ value, and compared this search results with those with using any avoidance regions. The fitness for the searches is defined to be a weighted sum of cost, power consumption and feasibility factor (with weights of 0.4, 0.3 and 0.3, respectively). The comparison results are depicted in Figure 8. Similar to the above experiments, the convergence rate for the constrained search is faster than that of the unconstrained case.

## 6 Discussion & Future Work

We have presented a framework for a designer to guide the search process in solving the hardware/software partitioning problem. The main contributions of our work include mapping a high dimensional fitness landscape to a three dimensional surface for easier visualization and interpretation. By using an appropriate scale factor, a 3-D fitness landscape can be readily generated and depicted. Furthermore, we show that such landscape information can be used by a designer to constrain the search space. The introduction of the integer X and Y coordinates for capturing a high-dimensional solution space greatly facilitates the specification and avoidance of low fitness regions.

The approach presented in this paper is quite general. First, as we pointed out previously, it is independent of the search mechanism used and is readily applicable to several different stochastic search techniques, e.g., evolutionary algorithms and simulated annealing algorithms. Secondly, in generating a fitness landscape for identifying bad solution regions, the designer can incorporate various design requirements. We have shown the use of a power consumption limitation and a feasibility factor bound

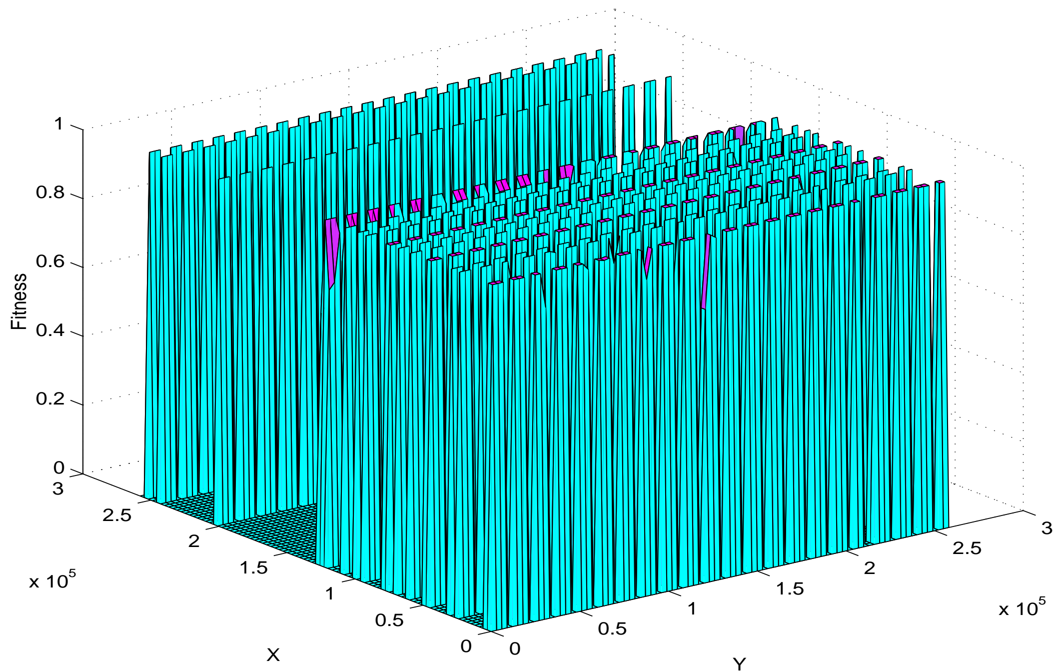


Figure 7: A fitness landscape for the feasibility factor attribute.

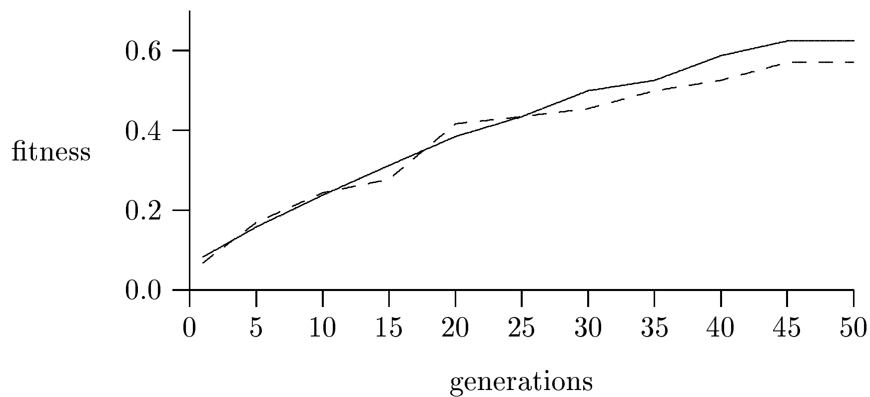


Figure 8: Fitness versus number of generations. Depicted results are averaged over 10 runs of **EvoC**, searching on the feasibility factor fitness landscape representation. The solid (dashed) line is for searches conducted with (without) constraints.

in the examples presented. In real applications, there may be other constraints to be considered, e.g., the compatibility among modules and availability of certain modules. All these constraints can be included in the generation of fitness landscapes. Moreover, there can be more than one landscape which capture different types of constraints. Eventually, all that a designer needs to do is to specify the "forbidden" regions in each landscape and let the search process quickly eliminate design candidates in these regions.

We have obtained quite encouraging results by using the landscape information to guide search in our partitioning tool **EvoC**. We are working on integrating the landscape generation process into **EvoC** and hence provide the user with a uniform environment for design space exploration. It will be interesting to use the landscape information in other search techniques. Furthermore, since the definition of neighbors plays a key role in delineating a fitness landscape, more in-depth study of neighbor relations can be very beneficial to the overall user-assisted design-exploration approach.

## References


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

[2] M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno. Hardware-software codesign of embedded systems. *IEEE Micro*, 14:26–36, 1994.

[3] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, 10:64–75, 1993.

[4] R. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10:29–40, 1993.

[5] E. Barros, W. Rosenstiel, and X. Xiong. A method for partitioning unity language to hardware and software. *Proc. European Design Automation Conf.*, pages 220–225, 1994.

[6] S. Prakash and A. Parker. Sos: Synthesis of application-specific heterogeneous multiprocessor systems. *J. Para. & Dist. Computers*, 16:338–351, 1992.

[7] S. Kumar, J. Aylor, B. Johnson, and W. Wulf. Object-oriented techniques in hardware design. *IEEE Computer*, 27:64–70, 1994.

[8] B. Dave, G. Lakshminarayana, and N. Jha. Cosyn: Hardware-software co-synthesis of embedded systems. *Proc. Design Automation Conf.*, pages 703–708, 1997.

[9] J. Teich, T. Blickle, and L. Thiele. An evolutionary approach to system-level synthesis. *Proc. Int'l Workshop Hardware/Software Codesign*, pages 167–171, 1997.

[10] R. Dick and N. Jha. Mogac: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems. *IEEE/ACM Int'l Conf. on CAD*, pages 522–529, 1997.

[11] L. Garber and D. Sims. In pursuit of hardware-software codesign. *IEEE Computer*, 31:12–14, 1998.

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

[13] J. D'Ambrosio and X. Hu. Configuration level hardware/software partitioning for real-time embedded systems. *Proc. of Third Int'l Workshop on hardware-software Codesign*, pages 34–41, 1994.

[14] S. Kauffman. *The Origins of Order.* Oxford University Press, NY, 1993.

[15] W. Chapman and J. Rozenblit. The system design problem is np-complete. *IEEE. Conf. Sys., Man, & Cyber.*, pages 1880–1884, 1994.

[16] Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Comp.*, 4:1–32, 1996.

[17] E. Weinberger. *J. Biol. Cybern.*, **63**:325, 1990.

[18] G. Greenwood and X. Hu. Are landscapes for constrained optimization problems statistically isotropic? *Physica Scripta*, **57**:22, 1998.

[19] Y. Saad and M. H. Schultz. *IEEE Trans. on Computers*, **37**:867–870, 1988.

[20] G. Greenwood and S. Ravichandran. Tech. Rpt. TR 98/01, Dept. of Computer Science, Western Michigan University, Kalamazoo, MI 49008 USA, 1998.

[21] R. Sambandam and X. Hu. Predicting timing behavior in architectural design exploration of real-time embedded systems. *Proceedings of the 34th IEEE/ACM Design Automation Conference*, pages 157–160, 1997.