Design Methodology for Development of Behavioral Synthesis Generic and Synthetic Benchmarks

Abstract

The importance of benchmarking as an essential methodology in evaluating and comparing digital systems synthesis tools is well established. Equally well are the limitations of the majority of current benchmarks recognized. Our primary goal is to lay out the theoretically and statistically sound foundations for addressing the key issues related to the selection of generic benchmarks and the synthesis and analysis of statistical synthetic benchmarks. The methodology for the synthetic benchmark generation uses as an intermediate step the methodology for the generic benchmark selection.

The benefits of our methodology for the generic benchmark selection are: (1) it provides the sound statistical foundations for the generic benchmark selection, (2) it establishes the fact that the development of a generic benchmark is a well-defined optimization problem. The advantages of the statistical synthetic benchmarks over the generic benchmarks include: (1) having an unlimited supply of synthetic design examples, (2) obtaining the real-life like design examples which suit a particular design or analysis need, (3) having a test set which better fits all available real-life design examples.

Due to the fact that the synthetic benchmark generation is based on the generic benchmark selection, the entire process of the generic and synthetic benchmark development is in sequence. It consists of the following five-stages: (1) collecting and analyzing real life design examples, (2) extracting and selecting the relevant properties that are used to specify designs, (3) deriving a generic benchmark set from the real life design examples, (4) developing the synthetic benchmark generator, and (5) constructing a synthetic benchmark set for the current applications or the future applications based on the trends of the development of applications.

The effectiveness of the proposed approach and the developed algorithms is demonstrated by the experimental design and validation of the generic benchmarks for the collected behavioral synthesis applications.

1. Introduction

1.1. Motivation

Benchmarking is widely recognized and used as a key methodology for the evaluation and comparison of competing products. However, often numerous objections are raised to many of commonly used benchmarks, mainly along the line that they are not designed to well reflect and test the real life industrial applications. In particular, synthetic benchmarks have questionable reputation.

Our strategic primary goal is to lay out the theoretically and statistically sound foundations for addressing the key issues related to the selection of generic benchmarks and the synthesis and analysis of statistical synthetic benchmarks.

Due to the fact that synthetic benchmarks must reflect real life design examples correctly, it has to be based on the generic benchmarks. Therefore, the synthetic benchmark generation requires to be preceded by the generic benchmark selection. We present the methodologies for the generic and synthetic benchmark development as a statistical optimization process. Our specific goal is to develop statistically validated behavioral synthesis generic and synthetic benchmarks.

1.2. Key Observations: What is a High Quality Benchmark?

Before discussing the qualifications of a high quality benchmark, we need to clarify what the applications of benchmarks would be. The most obvious ones are the comparison and selection of the given synthesis systems with the aid of benchmarks. We are interested not only in establishing proper ranking among the available synthesis tools, but also, more importantly, in identifying which synthesis system will most likely perform best for a given mix of customer applications.

Secondly, the information from a benchmark test can be used for the fine-tuning purposes during the development of synthesis software. Benchmarks provide a good guidance how to tune heuristics parameters in synthesis algorithms and compilation tools to the characteristics of typical real life design examples. This practice, however, as will be pointed out later, may cause an undesirable side effect called "over-tuning" unless we prudently use a set of "learning" benchmarks, which are different from the "testing" benchmarks. In that sense, this becomes a more interesting application of a synthetic benchmark since an unlimited number of "synthetic benchmarks" can be generated and applied to a synthesis system without taking the risk of "over-tuning" the system to a particular set of design examples.

Thirdly, benchmark test results can be used for software validation and verification purposes. Benchmarks are often an invaluable aid in "software system maintenance and debugging". If a benchmark test is performed whenever a software package needs changes, updates, or bug-fixes, one can be assured to a significantly higher degree that no new bugs are introduced by the changes.

Also, benchmark test results may be used to improve the quality of synthesis tools. To that account, a

benchmark test should be designed such that the results of the test give enough information to pinpoint a particular weakness of a particular tool. For example, if well designed, benchmark tests produce the information that characterizes the type of design examples on which a given tool does not perform as well as other tested tools do.

Finally, benchmark tests can be a useful research guidance. A well designed benchmark should help researchers to identify what are the key high-impact research areas and what design goals to optimize. For example, the comparison of the benchmark test results of currently available tools to the theoretical bounds of what can be achieved may give researchers insights as to which areas or optimization techniques should be investigated further in order to improve the currently available technology.

On the basis of the desired applications of benchmarks, we propose a set of quality criteria that will guide the design process of a benchmark: relevancy, compactness, comprehensiveness, and resolvability. The relevancy calls for a benchmark test to be based on real life design examples representing their full diversity and complexity. Since the implementation of design specifications and their syntheses are often time consuming and expensive, a benchmark test must contain as few designs as possible. At the same time, it is also important to cover all types of common designs. A good benchmark set strikes balance between its size and comprehensiveness based on real life design examples.

Finally, a benchmark should result in a fair comparison among competing tools and approaches with good resolutions. In other words, we do not want all the benchmarks too easy or too difficult, which may result in no differentiation among the various systems that are not equal in their quality.

A high quality synthetic benchmark needs to have one more qualification than a conventional set of benchmarks selected from real life designs do: the real life design space information must be incorporated into the synthetic benchmark set. The synthetic benchmark set is based on real life designs in the sense that it should "represents" them in their exact full diversity and complexity.

There are many potential advantages of "synthetic" benchmarks over "generic" ones. It is possible to generate as many design examples as required by the design process. For example, it can always produce a new series of synthetic design examples in an unpredictable manner so that a tool in question cannot be over-tuned to a particular design examples.

Design examples can be generated by a synthetic example generator with exact specifications we need. The specifications can include future trends, the various levels of difficulties, and the features of interest to a user. As we will show in this paper, the synthetic examples are often better representatives of real life design examples than those benchmarks derived directly from real life design examples do in the sense that it is more similar to all the members of a group it represents.

1.3. Key Ideas: How to Develop Generic and Synthetic Benchmarks?

Based on the observations presented in Subsection 1.2, we organize our ideas about designing generic and synthetic benchmarks. The generic and synthetic benchmarks are developed in five stages: (1) collecting

and analyzing real life design examples, (2) extracting and selecting relevant properties that are used to specify design examples, (3) deriving a generic benchmark set from the real life design examples based on the parameters describing the relevant properties of design examples, (4) developing the synthetic benchmark generator, (5) constructing a synthetic benchmark set for the current applications or the future applications based on the trends of the development of applications. Each step is explained in the reminder of the paper.

1.4. Paper Organization

The rest of the paper is organized in the following order. In Section 2, preliminary materials such as basic assumptions, definitions of terms, and design examples used in this paper are given. The related work is discussed in Section 3. After presenting the global design flow of the generic and synthetic benchmarks in Section 4, generic and synthetic benchmark design and optimization issues are discussed in Section 5 detailing the formulation of the problems solved in this paper and the explanation of the ideas behind the problem formulation. All the algorithms used in the synthetic benchmark generator are presented in Section 6. Sections 7 deals with experimental results. Finally, our conclusion is given in Section 8.

2. Preliminaries and Definitions

2.1. Definitions and Assumptions

All of the following terms "design," "design example," "design problem," or "synthesis problem" refer to an application example to be synthesized. By "benchmark," we mean an instance of the synthesis problem that will be tested upon the given set of products or, more specifically, synthesis systems. Sometimes, it also refers to a "benchmark test" or "benchmarking" meaning an application of a benchmark set upon the given set of synthesis systems. Generally, a "benchmark test" or "benchmarking" is a systematic and standardized methodology of evaluating and comparing competing products. A "benchmark set" refers to a set of design examples that are designed to achieve the goals of benchmark tests. A "generic" benchmark is a benchmark selected directly from real life design examples. A "synthetic" design example is an artificially generated instance of a synthesis problem which behaves like a real life design examples during synthesis.

The underlying computational model for synthetic design examples is the synchronous data flow graph (SDFG) on semi-infinite streams of data [LP95]. A design example is specified using a control data flow graph (CDFG) format [RP90].

Other underlying assumptions about the available design examples are the following. The available pool of design examples is reasonably complete in the sense that any new design example of interest is "similar" to at least one of the design examples already included in the pool of design examples in its diversity and in its distribution.

Two design examples are said to be "similar" if there is no big discrepancy between them in all the parameters relevant to the high level synthesis process. It is assumed that a pair of similar design examples test a set of similar functionalities and features of a synthesis system, therefore, eliminating the need to use both of them as benchmarks. Needless to say, the term "big" discrepancy used here is a relative one. The determining factors to dissimilarity are: (1) how differently a design example behaves, as compared with another one when it is tested on a high level synthesis system, and (2) how costly it would be to reduce the dissimilarity. The second point is related to increasing the benchmark set size. An obvious case is that all the available design examples are selected as benchmarks. Then the similarity of the benchmark set to the design example space is perfect at the cost of economy of benchmark tests.

2.2. Available Design Examples

In order to properly characterize the typical media design examples, we assembled 150 DSP, video, image, graphics, telecommunication, and information theory application examples. The assembled design set include three versions of the low pass 5th order elliptical wave digital filter, the 7th and 9th order wave digital filters, Winograd Convolution, several different forms (direct, cascade, parallel, continued fraction and Gray-Markel ladder) of the 8th order IIR bandpass Avenhaus filters, several different Winograd transformations, several 1D and 2D DCT algorithms, VHS transform, NTSC encoder and decoder, G2 blend surface calculation using quintic polynomials, several types of LMS adaptive filter, Hilbert Transform, 2nd order Volterra filter, 3rd order truncated Volterra filter, 28th, 50th, 60th, 100th, 126th and 133rd order FIR filters with a variety of transfer function and applied windowing techniques, several PCM DA converters, several NEC CD ROM digital to radio converters, image histogram, a variety of rank-order and linear video filters, Philips radio coder and decoder, Reed-Solomon, Goppa, Reed-Muller, Fire, Hamming and BCH codes.

Some interesting properties of the examples are: maximum sustained parallelism, percentage of nodes in cycles, critical path, integer iteration bound, most dominant pattern (multiply-add), second most dominant pattern (add-add), and coverage by the pair of dominant patterns. The maximum sustained parallelism for the examples ranges from 3 to 33 and the median is 8. The percentage of nodes in cycles are observed 0% to 100% with the median of 52%. The critical path lengths of the examples are from 3 to 3,161,852 with the median of 14. The percentage of the most dominant pattern and the second most dominant pattern are from 0% to 98% and from 0% to 46%, respectively. The median values of the percentage of the patterns are 39% and 13%, respectively. The coverage by the pair of dominant pattern ranges from 0% to 98% and the median is reported 39%.

2.3. Properties of Designs

We selected the following properties as being relevant to describing the designs. We first included the number of operations, the number of types of operations, the Gini coefficient for the number of operations. The Gini coefficient is an indicator of the concentration of one type of operations [Thi88]. The largest

possible value (100) indicates that all operations are of one type, the smallest possible value (0) indicates that all types of operations are equally represented. Timing measures are quantified through the following metrics: the ratio of the sample rate to the critical path, the number and types of nodes on the e-critical network (the subset of the graph which includes all nodes and edges on paths of the length greater than $(1 - e)$ times the critical path length, we used value $e = 0.2$), initial average and median scheduling slack, and the iteration bound. The concurrency measures used are the average parallelism, and the Gini coefficient for all types of operations, average and median variable density temporality [GPR94]. Finally, we also included regularity [GPR94] which captures the degree to which common patterns appear in the computation.

3. Short History of Benchmarking and Synthetic Benchmarking

It appears that the first benchmark in the computer architecture and compiler domains was the Gibson Mix [Gib70], which significantly influenced the design of IBM 360 and 370 series of computers. More recently, and in particular during the last several years, benchmarking was promoted in computer architecture as a key method for research and product evaluation. Small and partially concocted examples has been replaced with larger examples, which are manually analyzed for diversity [SPE90]. Since then, many key innovations in the architecture and compiler domains, such as cache memory, memory hierarchy, vector processing, RISC in general purpose computing, and multiplier-accumulator units, bit reversal units and circular buffers in general purpose programmable DSP processors, are direct consequences of attempts to achieve superior results on targeted benchmarks.

Before turning our attention to CAD benchmarking, it is worth mentioning the benchmarking of random number generators, where an exceptionally sophisticated and theoretically sound test system is developed [Knu81].

Recently, benchmarking attracted a good deal of attention in both research [Brg93] and industrial CAD communities. An analysis of DAC 1992 papers indicates that, while more than 30% percent of authors use their own benchmarks, more than 40% of the papers show results on "standard" benchmarks [Brg93]. Popular CAD benchmarks are available for sequential test generation, logic synthesis, physical design, circuit simulation [BBK89, Brg93], and DSP applications [ZMVSM94].

Although there are several often cited sets of high level synthesis benchmarks, the 5th order elliptical wave digital filter, by far the most popular benchmark, has been playing a more important role in the development of high level synthesis systems. For instance, a great number of new techniques (e.g., new scheduling and assignment algorithms and transformation mechanisms and) as well as many new optimization targets (e.g., testability, fault tolerance, power and latency minimization) were first illustrated (and sometimes even motivated and justified) by presenting the implementations of this example.

However, even a brief analysis of the 5th order elliptical WDF reveals many unpleasantly surprising facts. The example was first used by Dewilde [DDN85], and in 1988 the IMEC group presented the full implementation of this example according to CCITT standards [C+88]. They noted that the structure

is unnecessarily complex for the required transfer function.

Sometimes functionally incorrect implementations were presented. The instances of the 5th order elliptical WDF where some operations were simply deleted have been published. Some authors simply remove feedback edges, which does not result only in functionally incorrect implementations but also in significantly reducing both the number of required interconnect and registers, giving them very improper cost advantages. Although all multiplications are by constants, and de facto standard procedure is to use expensive multipliers for implementations, the available time is set unrealistically low (equal or very close to the critical path).

The peculiarity of properties of the 5th order elliptical WDF is equally remarkable. For example, a study of several hundred DSP examples indicated that the 5th order elliptical WDF is the only example where all nodes are in cycles and where the critical path was from algorithmic delay to algorithmic delay. Furthermore, this benchmark example has regularity significantly lower than the great majority of DSP examples and exceptionally high non-uniformity of distribution of operations over time. Since wave digital filter are widely recognized as exceptionally numerically stable, it is exceptionally sensitive on the application of computational transformations.

4. Approach: Generic and Synthetic Benchmark Design Flow

The global design flow of the synthetic benchmark is shown in Figure 1. The shaded area of the diagram indicates the selection process of generic benchmarks. The generic benchmark selection process starts with collecting and analyzing real life design examples. The analysis is aimed at obtaining a set of relevant parameters that describes properties of a CDFG. The set of parameters should be able to describe the full diversity and complexity of a given application domain.

Once the parameter set is finalized, we use a simulated annealing-based algorithm to select a generic benchmark set. The generic benchmark set is validated using the "substitution" technique [ET93].

The generic benchmark set serves as a starting point from which the parameters of synthetic benchmark set are defined. Since, as explained in Section 5, each generic benchmark represents a group of similar real life design examples, it is possible to identify those groups of design examples by measuring the similarity of each design example to each benchmark.

The next step in synthetic benchmark design is to develop a synthetic example generator (SEG). The SEG generates random design examples given a set of characterizing parameters observed in the prior steps. The formal formulation of the random example generation problem is given in Section 5. The algorithm for SEG is presented in Section 6.

Once the SEG is developed, a synthetic benchmark set based on the generic one can be generated. Furthermore, by extrapolating the current trend in an application domain, we can easily obtain statistically predicted future applications to test to see how well a synthesis tool can treat the future designs.

After synthetic design examples are generated, they are validated using statistical techniques.

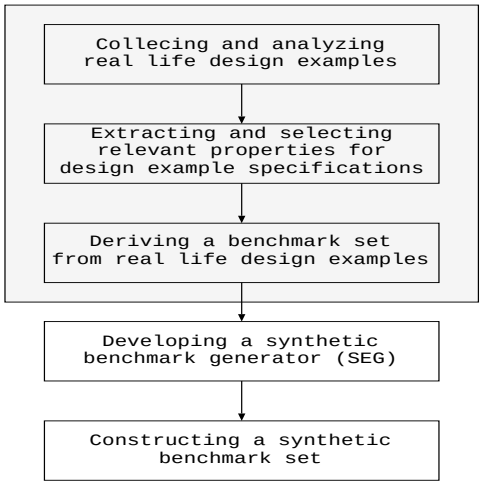

Figure 1. The global design flow of the synthetic benchmark generation

5. Generic and Synthetic Benchmark: Design and Optimization Issues

5.1. Generic Benchmark and Parameters for Synthetic Benchmark

Selecting a set of generic benchmarks from the real life design example set is a step further to the generation of synthetic benchmarks. The selection can be done by identifying clusters in the design example space and selecting a set of design examples that best characterizes the design example space. For example, as depicted in Figure 2, a set of design examples are distributed in a design example space. There are two discernible clusters in the design example space. There are such a pair of design examples, indicated by A and B, that in some sense better represent the two clusters than any other pair of two examples.

The generic benchmark selection problem may be informally described as follows:

"Given the size of a generic benchmark set, find a set of design examples that best represents the given design example space."

More formally,

"Given the size of a generic benchmark set n and a set of m examples $E = \{e_i | i = 1, 2, ..., m\}$, find a set of examples $C = \{c_j | j = 1, 2, ..., n\}$ such that

Minimize :

$$D_n = \sum_{k=1}^{|R|} d_k, \tag{1}$$

where $R = \{r_k | r_k \in E - C, k = 1, 2, ..., m - n\}$, $d_k = \min\{d(r_k, c_j) | j = 1, 2, ..., n\}$, and $d(r_k, c_j)$ refers to a distance function from r_k to c_j."

Identification of the size of a generic benchmark set is done by the "characterization" process. The process involves the iterative tests that examines the effect of selecting a certain size by measuring total distances from each design example to selected examples as given by the expression (1). We continue the set size determination process until we reach a point where the set size strikes balance between the economy of benchmarks and its effectiveness. We compute and compare the total distances incrementing the size one at a time until we get no significant improvement by adding more design examples to the benchmark set.

The generic benchmark set size problem is formally given by:

"Given a set of m examples $E = \{e_i | i = 1, 2, ..., m\}$, select the generic benchmark set size n such that:

$$\min\{n | r \leq \frac{D_{n+1}}{D_n}, n = 1, 2, ...m - 1\}, \tag{2}$$

where D_n is given by the expression (1) and r refers to a target ratio."

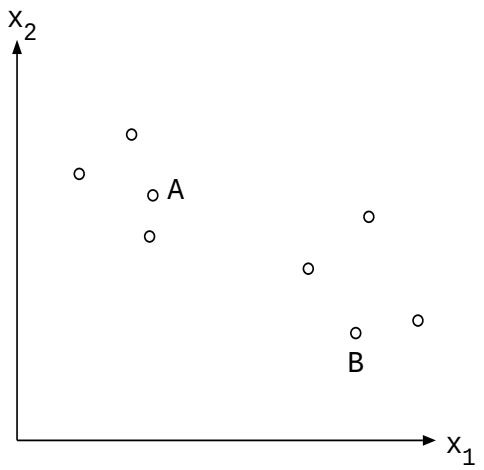

Figure 2. A design example space and candidate benchmarks (indicated by A and B)

The generic benchmark set is validated using the resubstitution technique [ET93]. A number of randomly chosen design examples are eliminated from the pool of design examples from which the original benchmark set is selected. A new benchmark set is selected. By comparing the new benchmark set with the original one by the "weighted bipartite matching" technique, we can measure the validity of the original and the new ones. We use *the Hungarian Method* [Kuh55] to solve the problem.

From the generic benchmark set, we define a set of parameters that are to be used to generate a set of synthetic examples. As stated earlier, since the synthetic benchmarks are to be more similar to all the members in the group it belongs to, we generate a set of parameter that specifies an imaginary design

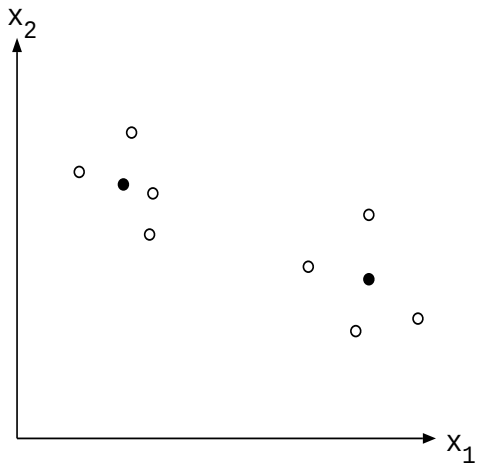

Figure 3. "Ideal" synthetic benchmarks for the design example space

example by computing the mean values of all dimensions of all the design examples in the group. The group is identified by measuring the distances from generic benchmarks to the other design examples. The cluster identification problem is formally given by:

"Given a set of m design examples $E = \{e_i | i = 1, 2, ..., m\}$, and a set of generic benchmarks $C = \{c_j | j = 1, 2, ..., n\}$, find clusters of design examples $G = \{\{g_i\} | i = 1, 2, ..., n, \sum_k^n |g_k| = m - n\}$ such that:

$$g_i = \{r_j | d(c_i, r_j) : minimal, j = 1, 2, ..., m - n\}, \tag{3}$$

where $r_j \in R$, $R = \{r_j | r_j \in E - C, j = 1, 2, ..., m - n\}$, and c_i is the i^{th} generic benchmark."

5.2. Synthetic Example Generator

The generation of a random graph obeying a set of given parameters is not computationally tractable. Therefore, we have to resort to adopting a heuristic method.

Informally, the synthetic example generator (SEG) problem can be described as follows:

"A set of quantifiable parameters relevant to characterizing a CDFG and a set of component numbers are given. Our objective is to generate an individual synthetic CDFG such that the individual synthetic CDFG conforms to the desired characteristics imposed by the given set of parameters."

The SEG problem can be defined in more formal terms:

"Given an n-tuple of quantifiable parameters $(p_1, p_2, ..., p_n)$ and an m-tuple consisting of quantities of available CDFG components $(c_1, c_2, ..., c_m)$, generate a synthetic CDFG which contains exact numbers

of components specified by the m-tuple of component quantities and have properties as close to those described by the n-tuple of quantifiable parameters as possible."

6. Synthetic Benchmark: Optimization and Statistical Validation Algorithms

6.1. CDFG Representation

Figure 4 shows an example of conceptual representation of a CDFG which is used in constructive generation of a synthetic CDFG. The smaller circles in the figure indicate primary inputs and outputs while larger circles represent various arithmetic operations. Rectangles with one side open are delays. Note that the set of delays on the top of the figure is exactly identical to the set of delays at the bottom of the figure. It shows all the primary inputs and delay outputs on the top of the figure and all the primary outputs and delay inputs at the bottom. Since an iteration starts with firing nodes with all the required inputs from delays or primary inputs and ends with all the primary outputs and delay inputs, our representation of CDFG clearly shows the iteration boundary of a CDFG. For that reason, we get additional advantages such as that it is easy to reason about the length of the critical path, the presence of cycles, and the presence and length of loops.

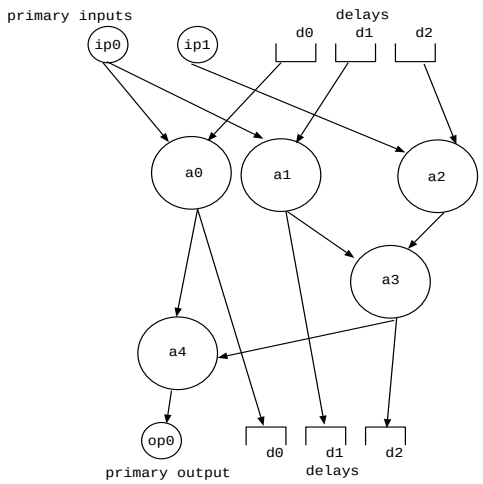

Figure 4. A CDFG representation showing its iteration boundary

The identical CDFG in usual representation is given in Figure 5.

6.2. Observations

Since we are given a set of components and are supposed to generate an SCDFG using the set of components, the problem reduces to placing edges between pairs of components. There are a number of

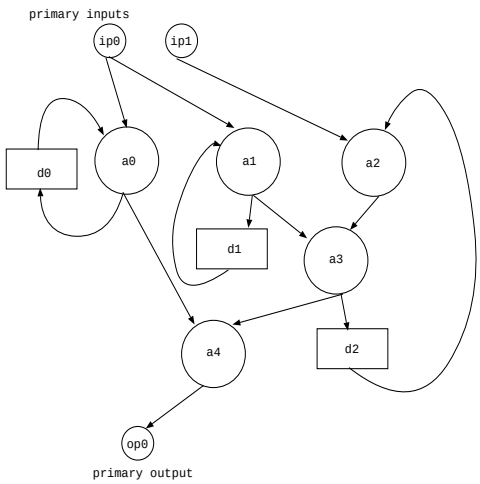

Figure 5. The usual representation of a CDFG identical to the graph in the previous figure

restrictions in placing edges in a CDFG. The following observations give a summary of the restrictions in constructing a random SCDFG.

1. All the primary inputs and all the outputs from components should be used as an input to one or more components (arithmetic operations, delays, primary outputs and conditionals). A primary input cannot be connected to a primary output. The output from a component should not be connected to more than one primary output.

2. A primary output should be connected to one and only one component.

3. A primary input or an output of a component cannot be connected to the output end of a component.

4. A primary input or an output of a component cannot be connected to any of primary inputs.

5. A delay component can have one and only one input.

6. All the binary arithmetic operation components must have two input connections. Likewise, all the unary operation components (e.g., shifts) should have only one input edge.

7. A delay component can be placed in-between any pair of components (even in-between a pair of two delays).

8. A loop L can be formed by connecting the input end of a delay component D to the output end of a component C and the output end of D to the input end of a component that is chained to the component C.

9. The critical path is determined by the longest possible path from primary inputs and delays to primary outputs and delays. The longest path can be computed by using dynamic programming and recursive function calls.

6.3. Construction of an Initial Random SCDFG

The generation of a synthetic CDFG consists of two stages. First, a random SCDFG without imposing the given property parameters is constructed. Then, we iteratively swap two edges so as to make the properties of the SCDFG close to the targeted. By "edge-swapping," we mean the exchange of destinations of two edges.

The top-most algorithm for the SEG is given in Figure 6.

```
synthetic_CDFG ( ) {
   initial_graph ( );
   simulated_annealing ( );
}
```

Figure 6. SEG algorithm

The initial random SCDFG is constructed in four steps. First of all, to ensure the observation 1 given in Subsection 6.2 is not violated, all primary inputs are connected to a randomly chosen set of components. The next step is to follow the observation 2 by using the algorithm given in Figure 11. The rest two steps shown in Figure 12 and 13 involve the observation 1, 5, 6, and 7. Note that by stepping through the above orders, we ensure that the observations 3, 4 are not violated.

```
initial_graph ( ) {
   connect_input_edges ( );
   connect_output_edges ( );
   connect_all_outputs ( );
   connect_all_inputs ( );
}
```

Figure 7. Initial random CDFG construction algorithm

There are several conditions to be met when selecting the number of edges originating from primary inputs and delay components. First, there should be at least one arithmetic operation that has all the required inputs from primary inputs or delay components. Second, besides such an arithmetic operation described above, there should be at least one arithmetic component that has all the required inputs but one from primary inputs or delay components. These two requirements are needed to ensure there is no "inevitable" cycle in a SCDFG. Also, all the primary inputs and delay components should be connected

to one or more arithmetic components. Furthermore, the number of edges from primary inputs and delay components cannot be greater than the sum of all required inputs of all operations and should be less than the sum of all required inputs of all operations plus the number of primary outputs minus the number of arithmetic and logical operations. The latter two conditions are necessary to avoid the cases where the observation 1 in Subsection 6.2 is violated.

```
connect_input_edges ( ) {
   select_number_of_input_edges ( );
   connect_primary_inputs_to_nodes ( );
}
```

Figure 8. Algorithm to place outgoing edges from primary inputs

The number of edges from primary inputs is given by:

$$\max(2 \times \min_j(i(n_j)) - 1, p) \leq n(e_{pi}) \leq \min(\sum_j(i(n_j)) - k + q, \sum_j(i(n_j))), \quad (4)$$

where $n(e_{pi})$ = the number of edges from the set of primary inputs, p = the sum of numbers of primary inputs and delays, k = the number of components with outputs, q = the sum of numbers of primary outputs and delays, and $(i(n_j))$ = the number of required inputs of the component j. The algorithm is given in Figure 9.

```
select_number_of_input_edges ( ) {
   generate a random number such that:
      (1) there is at least one node that has a complete set of
          inputs connected to either primary inputs or delays
      (2) there is at least one node that has at least a subset
          of required inputs from either primary inputs or delays,
      (3) the number should not be greater than all the
          required inputs of all the nodes minus the number of
          nodes plus the number of primary outputs and delays, and
      (4) the number should be greater than or equal to the number
          of primary inputs and delays.
}
```

Figure 9. Computing the number of edges from primary inputs and delays

After randomly selecting a number satisfying Inequality (4), edges are to be actually placed. According to the observation 1, each primary input has to be connected to at least one arithmetic or delay component. Therefore, we first put an edge from each and all the primary input to a randomly selected arithmetic operation or a delay component. If there are more edges left to be placed, we randomly choose pairs of

a primary input and an arithmetic operation or a delay component to place edges between them. The edge placement algorithm is given in Figure 10.

```
connect_primary_inputs_to_nodes ( ) {
   place edges from primary inputs and delays to operations such that:
      (1) all inputs and delays have at least one edge incident to them, and
      (2) at least one node has a complete set of inputs
}
```

Figure 10. Placing edges from primary inputs and delays

Next, all primary outputs and all delay components get their respective inputs. This is done by randomly selecting a node in a CDFG other than primary inputs and connecting it to a primary output or a delay component in sequence. The conditions to be met are, as shown in Figure 11, (1) each primary output or delay can have only one incoming edge, and (2) no more than one primary output or delays can have incoming edges originating from the same node.

```
connect_output_edges ( ) {
   place edges from operations to primary outputs and delays such that:
      (1) there is one and only one edge connecting a operation and
          a primary output or a delay, and
      (2) there is no edges connecting from the same operation to
          more than one primary outputs.
}
```

Figure 11. Placing incoming edges to primary outputs and delays

The rest of the necessary edges to complete the initial construction of CDFG are placed in the next two steps. They consist of connecting all the outputs of nodes and completing all the necessary input edges of all nodes. The algorithms are given in Figure 12 and Figure 13, respectively.

```
connect_all_outputs ( ) {
   place edges from an output of an operation to
   an input of another operation such that:
      (1) there is no cycles, and
      (2) the outputs of all the operations should be used.
}
```

Figure 12. Placing outgoing edges from all operations

```
connect_all_inputs ( ) {
   place edges from an output of an operation to
   an input of another operation such that:
      (1) there is no cycles, and
      (2) the inputs of all the operations should be filled.
}
```

Figure 13. Placing incoming edges to all operations

6.4. Iterative Improvement of the Initial Random SCDFG

We apply the simulated annealing [KGV83] to the randomly constructed SCDFG to obtain an SCDFG with the given properties. The simulated annealing process starts with generating a random initial proposal. In our case, the initial SCDFG constructed in the first step serves as the initial random proposal. Then all the required objective functions are evaluated to measure the nearness of the proposal to the target solution. If it does not meet the requirements, it enters iteration cycles which stop either when the objectives are met or when a pre-specified number of iterations are performed.

The simulated annealing iterations are composed of three major parts: generating a new random proposal by applying a controlled perturbation technique on the old proposal, evaluating the objective functions, and accepting or rejecting the new proposal.

By making a number of controlled random modifications to the old proposal, we get a new proposal which may or may not be better. If the new proposal is better than the old, we accept it. Otherwise, we accept it with a probability that is determined by its closeness to the best solution ever found and the current temperature. The temperature controls the iteration and the acceptance ratio of worse solutions at each iteration. The temperature is reduced after trying new solutions pre-specified times. When the temperature reaches to zero, the iteration stops regardless of the status of the best solution found.

The crucial tasks, then, are to find: (1) the way a new solution is constructed and (2) the acceptance function when a new solution is worse than the old. As said earlier, the "next" solution (SCDFG) to be compared to the "current" solution (SCDFG) is computed by swapping two randomly chosen edges. Clearly, by doing this, the new SCDFG has a new properties (e.g., the length of the critical path, the length of the longest loop, etc.). Our acceptance function for worse solutions is based on the exponential distribution determined by the temperature and the closeness of the new solution to the best solution ever found.

This process is performed until the SCDFG has properties defined by the set of parameters or some number of iterations are done (in this case when the temperature variable that controls the acceptance ratio and iterations is zero). We use standard geometric cooling schedule.

7. Experimental Evaluation

We collected and analyzed the design examples presented in Section 2.2. Figure 14 shows the generic benchmarks selected using the procedure described in this paper.

1. NEC digital to analog converter
2. Motorola's 126-tap FIR filter
3. IMEC modem
4. 7th order WDF filter
5. 8 point 1 dimension decimation-in-time DCT
6. 5th order distillation plant controller
7. 2nd order Volterra filter
8. (31, 15)BCH code
9. 2D 8 point VHS transform
10. Hilbert transform
11. 2-D 10-th order IIR filter
12. 9-order rank-order filter.

Figure 14. The generic benchmark set

Table 1 shows an application of the resubstitution technique to evaluate the validity of the selected benchmark set. A set of 150 design examples is used to obtain the initial benchmark set. We compute a new benchmark set after randomly eliminating 25 design examples from the design pool. At each trial, we observe the number of benchmarks that are replaced by new ones and the difference between two sets of benchmarks. For the computation of the difference between two sets of benchmarks, we use the *Hungarian Method* [Kuh55].

	Min. Number of different benchmarks	Max. Number of different benchmarks	Avg. Number of different benchmarks
Resubstitution 1 (size: 125)	0	0	0
Resubstitution 2 (size: 100)	0	2	0.7
Resubstitution 3 (size: 75)	0	3	1.7
Resubstitution 4 (size: 50)	2	6	3.2

Table 1. The results of resubstitution (original set size: 150)

We run this sequence of tests 10 times and the results are summarized in Table 1. The second and third columns are for the minimum and maximum number of replaced benchmarks from the original benchmark set. In the final column, the average number of benchmarks replaced is given. As the result clearly shows there is no big difference in the configuration of benchmarks when resubstitution is applied.

In addition, even though there are cases where several benchmarks are replaced by some other design examples, the difference in terms of distance between two set of benchmarks are small enough to claim that those sets are essentially similar. When a benchmark set obtained by the resubstitution technique is compared to the original benchmark set by way of measuring the total distances from the set to all the design examples in the original set, the difference in terms of distance is less than 4%. This indicates the method presented in this paper is highly effective [ET93].

Table 2 shows a set of experimental data that are run on 5 different sets of design examples. Each number in the columns "Original Benchmark" and "Synthetic Benchmark" indicates the sum of distances from benchmark designs to the rest of the designs measured in the L_1 (*Manhattan*) distance. The improvement ratio is given in the "Improvement" column. The running time for each case of computing the benchmark set and corresponding synthetic benchmark set is given in the right most column.

Size of Design Example Set	Size of Benchmark Set	Original Benchmark	Synthetic Benchmark	Improvement	Running Time (sec.)
25	5	4551	3649	20%	8
50	10	7765	6543	16%	31
75	15	902	730	19%	51
100	10	1282	1122	13%	99
125	10	22927	21234	7%	89
150	10	2262	2006	11%	109

Table 2. The sum of distances from corresponding benchmarks to design examples

In Table 3, results of SCDFG generation experiments are shown. In the experiments, we observe the values of the properties with the highest discrepancy between the targeted and the achieved given the number of operations. Each row shows the initial and final values of such a property in "Initial" and "Final" columns, respectively. The initial and final refer to the values of the property of SCDFGs before and after an application of the simulated annealing-based algorithm, respectively. The last two columns show the discrepancies between the target values and the final values.

8. Conclusion

In this paper, we discuss the design and validation of the generic benchmarks and the synthetic CDFG generator. The methodology for the synthetic benchmark generation uses as an intermediate step the methodology for the generic benchmark selection.

The benefits of our methodology for the generic benchmark selection are: (1) it provides the sound statistical foundations for the generic benchmark selection, (2) it establishes the fact that the development of a generic benchmark is a well-defined optimization problem. The advantages of the statistical synthetic benchmarks over the generic benchmarks include: (1) having an unlimited supply of synthetic design examples, (2) obtaining real-life like design examples which suit a particular design or analysis need, (3)

No. of Operations	Target	Initial	Final	Discrepancy	Discrepancy / Target (%)
55	20439	35723	20517	78	0.4
65	15334	27545	15329	-5	0.0
130	28565	46921	27524	-1041	3.6
220	45977	66372	43862	-2115	4.6
355	64261	127544	67387	+3126	4.9
355	59168	81661	59165	3	0.0
355	52023	84664	56143	+4120	7.9
950	178554	466142	178591	+37	0.0
950	166269	489667	170046	+3777	2.3

Table 3. The results of SCDFG generation

having a test set which better fits all available real-life design examples.

Due to the fact that the synthetic benchmark generation is based on the generic benchmark selection, the entire process of the generic and synthetic benchmark development is in sequence. It consists of the following five-stages: (1) collecting and analyzing real life design examples, (2) extracting and selecting relevant properties that are used to specify designs, (3) deriving a generic benchmark set from the real life design examples, (4) developing the synthetic benchmark generator, and (5) constructing a synthetic benchmark set for the current applications or the future applications based on the trends of the development of applications.

The effectiveness of the proposed approach and the developed algorithms is demonstrated by the experimental design and validation of the generic benchmarks for the collected behavioral synthesis applications.

References

[BBK89] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. *1989 Int. Symp. Circuits and Systems*, pages 1929–1934, 1989.

[Brg93] F. Brglez. ACM/SIGDA design automation benchmarks: Catalyst or anathema? *IEEE Design and Test of Computers*, pages 87–91, September 1993.

[C+88] L. Claesen et al. Automatic synthesis of signal processing benchmark using the CATHEDRAL silicon compilers. *Custom Integrated Circuits Conference*, 1988.

[DDN85] P. Dewilde, E. Deprettere, and R. Nouta. Parallel and pipelined implementation of signal processing algorithms. In S. Y. Kung, H. J. Whitehouse, and T. Kailath, editors, *VLSI and Modern Signal Processing*, pages 258–264. Prentice Hall, Englewood Cliffs, NJ, 1985.

[ET93] B. Efron and R.J. Tibshirani. *An introduction to the bootstrap*. Chapman and Hall, 1993.

[Gib70] J. C. Gibson. The Gibson Mix. Technical Report Report No. TR 00.2043, IBM Systems Development Division, Poughkeepsie, NY, 1970. Widely quoted as a research done in 1959.

[GPR94] L. Guerra, M. Potkonjak, and J. Rabaey. Concurrency characteristics in DSP programs. In *Proceedings of ICASSP-94. IEEE International Conference on Acoustics, Speech and Signal Processing*, 1994.

[KGV83] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 1983.

[Knu81] Donald Ervin Knuth. *The art of computer programming.* Addison-Wesley series in computer science and information processing. Addison-Wesley, Reading, Mass., 2nd edition, 1981.

[Kuh55] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quaterly*, 2:83–97, 1955.

[LP95] E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. of IEEE*, 83(5):773–799, 1995.

[RP90] J. Rabaey and M. Potkonjak. Resource driven synthesis in the hyper system. In *1990 IEEE International Symposium on Circuits and Systems*, volume 4, New Orleans, LA, USA, May 1990.

[SPE90] SPEC Newsletter. *SPEC Benchmark Suite Release 1.0*, volume Vol. 2, No. 2, Fremont, CA, 1990. Waterside Associates.

[Thi88] R. A. Thisted. *Elements of statistical computing.* Chapman and Hall, 1988.

[ZMVSM94] V. Zivojnovic, J. Martinez Velarde, C. Schlager, and M. Meyr. DSPstone: a DSP-oriented benchmarking methodology. In *Proceedings of the 5th International Conference on Signal Processing Applications and Technology*, 1994.