DNA Based Evolutionary Approach for Microprocessor Design Automation

N Venkateswaran[1], Arjun Kumeresh M[2], Harish Chandran[2]

[1]Director, WAran Research FoundaTion (WARFT), Chennai, India
[2]Research Trainees WARFT (in alphabetical order)
{waran, arjun, harish}@warftindia.org

Abstract. In a paper [1] presented to BICS 2006, a basic methodology for microprocessor design automation using DNA sequences was proposed. A refined methodology with new schemes for traversal, encoding, recombination, and processor evaluation are proposed in this paper. Moreover concepts such as mutation, graphical decoding and environment simulation are introduced and a new technique for creating DNA based algorithms used in the mutation process is also presented. The proposed methodology is then generalized to extend its application to other domains. This paper presents a conceptual framework whose implementation aspects are still under investigation.

1 Introduction

Conventional microprocessor design automation involves optimization at various phases to minimize gate count, power, chip area and to maximize performance. Many tools are available to automate these processes. Evolutionary algorithms are applied towards these optimization processes. This is due to the fact that evolution is an optimization process by which the phenotype of a population gets optimized over successive generations. However, the functional level architectural design is not a part of the automation. Rather the architecture is created by a team of architects using their prior experience of designing various microprocessors. The fact that evolution has produced complex organisms such as humans stands testimony to its success as an optimization process. Natural evolution involves DNA based processes. But modeling this natural evolution at the DNA level with realistic encoding and recombination processes has not been attempted to automate microprocessor design [2]. In fact, microprocessors, which are less complex than humans, can be evolved naturally without human intervention if their characteristics (phenotype) can be mapped onto the DNA domain through a biologically realistic encoding process. Then automating microprocessor design process would reduce to combining different microprocessor DNAs to produce an offspring and simulating their working environment thereby incorporating random variation and selection which form the two major steps in natural evolution. This paper proposes such a DNA based approach for automating microprocessor design (for both general purpose as well as ASIC) which involves automation at all levels from the functional level to the layout level.

2 The Methodology

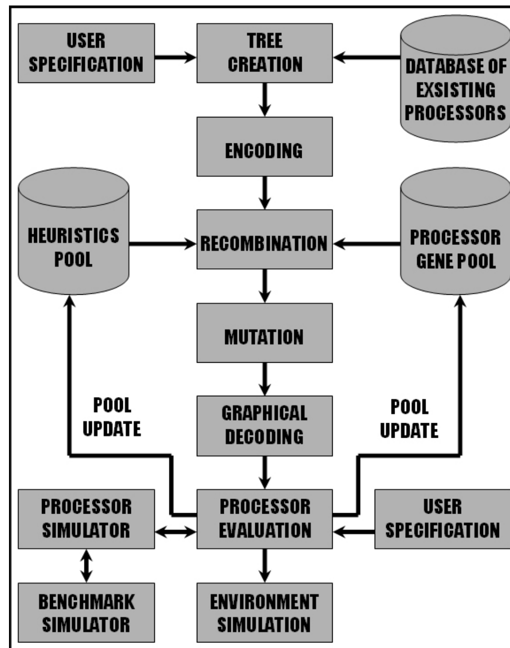

Fig. 1. The Proposed Methodology

Fig. 1 shows the proposed methodology. Specifications of the desired processor (***User Specifications***) are obtained from the user. The user can mention the power and performance ratings required from the processor and also its functionalities. A typical spec could be: 32 bit ALU based General Purpose processor rated at 3 Gigaflops at 80 Watts. An example of an ASIC spec could be: 32 bit processor with high fault tolerance, for image processing applications performing at 2 Gigaflops at 20 Watts. A ***Database of Existing Processors*** is built up beforehand. If an appropriate match is found, then that design is given to the user. Else, processors having similar characteristics are chosen and their ***Parameter Trees*** are built.

'Characteristics' of a processor refer to those parameters which completely and uniquely describe a processor. These parameters, such as clock speed, netlist density, threshold voltage, on chip memory etc., play a vital role in the design and working of the processor. The DNA sequences of the processors are then obtained using a specific, nature inspired ***Encoding*** scheme. ***Recombinations*** between various sequences are driven by certain heuristics. Certain components of a processor like Carry Save Adder, Baugh-Wooley Multiplier etc are pre-encoded and stored in the ***Gene Pool***. These components can then be added to the offspring processor based on the user specifications. Moreover, processors evolved in previous iterations are also stored in the gene pool. ***Mutation or Post Processing Stage I*** involves the extraction of the overall netlist of the processor. ***Graphical Decoding*** is done to fix the layout level map of the processor and incorporates ***Embryogenesis***. Next, the processor is evaluated using a ***Processor Simulator*** that runs a ***Benchmarking Suite*** on it. Minor deviations from the user specifications are corrected in ***Environment Simulation*** or ***Post Processing Stage II***. Here, environment refers to a set of applications that are to be run on the processor. Finally, after the processor is tested and evaluated, the goodness of each heuristic used in its creation is updated. If a satisfactory processor is found, then the gene pool is also updated with that processor. The following sections elaborate on these processes.

3 Tree Creation & Traversal

The first step in the methodology involves mapping the characteristics of a processor onto a ***Directed Acyclic Graph*** (parameter tree) where each node represents a characteristic and the edge weight between two nodes quantifies the dependency between the nodes.

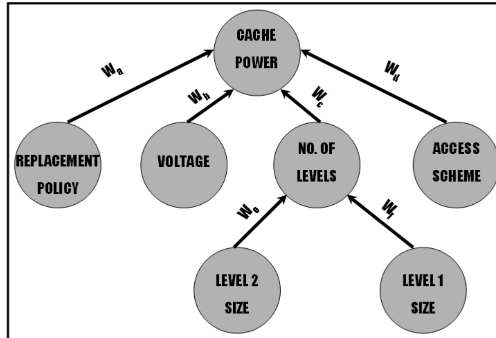

Fig. 2. Example: Part of a Cache Parameter Tree

Fig. 2 shows a part of the parameter tree of a cache structure. According to this tree, the power consumed by the cache depends on 4 parameters. Of these, ***No. Of Levels*** has 2 children. This is a direct consequence of the fact that the values of the parameter ***Level 1 Size*** and ***Level 2 Size*** is not 0. The edges in the figure have weights W_a, W_b … W_f.

These weights signify the level of dependence between two nodes: higher the weight, higher the dependence. For example, ***Cache Power*** is most affected by ***Voltage***. Hence W_b will be greater than W_a, W_c and W_d. These weights play a vital role while traversing the tree.

To build the parameter tree, various characteristics that define a processor must be obtained. Then various relationships among these characteristics must be studied. This can be done by collecting a large number of processor specifications and observing the trend among various parameters. For example, if most of the processors show that increasing the cache levels increases the power consumption tremendously, a high edge weight can be associated between these two parameters. A simple data mining algorithm can automate this process.

The Node Function of a particular node is a mathematical relation connecting the ***Node Data*** of the node to the ***Node Data*** of its parents and children. Fig. 3 shows a simple functional dependence. The parameter MNIPC (Maximum Number of Instruction Per Clock-cycle) is dependent on 3 of its parents: the No. of Cores, the NDP (Number of Data Paths) per core and the Number of Processors. MNIPC is just the direct product of these values. The node function is very important because it defines the actual mathematical relations between nodes. Thus node values can be fixed up and validated during recombination. For example, if the user specifies a MNIPC of 6, then we can fix the No. of Cores, No. of Processors and NDP as any combination of 1, 2, 3 (1x2x3=6). This could be one of the heuristics during recombination.

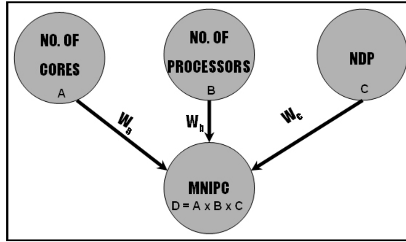

Fig. 3. The Node Function

Fixing these functions can be quite difficult as complete knowledge of processor architecture is required to manually find the functions. But it is possible to automate this process of finding the node function using supervised learning algorithms. For this, large number of processors must be analyzed and their trees must be created.

By applying this process to non connected nodes, inferences can be drawn about the way they interact indirectly. For example, instruction size and type of full adder used in the ALU don't have any obvious relationship. But if hidden relations do exist between them, then they could be found out using this method. The node function can be stored in standard formats like post-fix notations with variables as pointers to other nodes. The node function is mostly architecture independent. But, sometimes, certain architectures may change the function to an extent. For example, the inclusion of ***Hyper-threading*** may change the equation governing ***Level 1 Cache Size***.

A ***Depth First Traversal*** (DFT) is performed on the parameter tree to build a linear sequence of nodes such that related nodes are placed as close to each other as possible. In natural DNA sequences, related genes are placed close to each other. Since DFT visits a node that is most strongly related to the node being processed, it provides a list where related nodes are placed together. By specifying directions on the edges, the DFT is restricted in exploring only the children of the node being processed. When a node has multiple parents and has a stronger connection to one of its unexplored parents than any of its children (like node G in Fig. 4), it would be obviously better to place that parent node next to the node being processed.

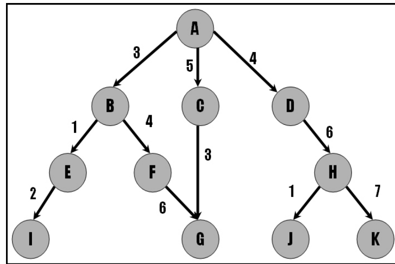

Fig. 4. Traversal Example

To facilitate this, the ***Adjacency Matrix*** of the parameter tree is symmetrized. This allows the DFT to move to any node connected to the node being processed. This technique is called ***Bi-Directional Depth First Traversal*** (BDFT). Finally, a ***Depth Limited Traversal*** would ensure that a single branch does not monopolize the optimal placement condition.

Thus the path taken by the DFT algorithm on this graph is ***A → C → G → D → H → K → J → B → F → I → E*** while path taken by BDFT is ***A → C → G → F → B → D → H → K → J → E → I***. To estimate the placement in the list, a metric known as ***Placement Factor*** is computed. The placement factor gives a measure of how well a node is placed in the list with respect to its distance from related nodes. Obviously, a

uniform placement factor for all nodes is desirable. This would mean that every node is equally well placed in the list.

The equation of placement factor of Node i is given by:

$$\text{Placement Factor (Node i)} = \sum (W_{ij} \times D_{ij}) / \sum (W_{ij}) . \quad \mathbf{(1)}$$

with summation over j and where W_{ij} is the weight between Node i and Node j and D_{ij} is the distance between Node i and Node j in the list

Smaller values of the placement factor indicate that the placement of that particular node is good. Initially the average placement factor is computed. Then all nodes having values greater than the average are arranged so as to reduce their placement factor value. This arrangement is done one node at a time starting from the worst placed node. When the placement factor of that node is reduced below the average, the placement factors for all nodes are recomputed and the entire process repeated. To avoid oscillations, a history of changes is kept. Any recurring pattern is identified and the responsible nodes are blacklisted. Thus the overall effectiveness of placement of the nodes in the list is improved.

4 Encoding

The process of creation of the actual DNA strands is carried out in the encoding phase. A DNA sequence that totally describes a processor on decoding is bound to be complicated. Such complicated sequences will be difficult to work with during recombinations unless they are well formed and ordered. Thus a good encoding scheme must be biologically realistic, yet well ordered. By placing related nodes near each other, we allow ***Localization of Reactions*** wherein related characteristics are changed simultaneously. This bio inspired node placement scheme provides a good platform to build simple recombination rules. The encoding process transforms information to a base 4 domain. The base 4 system allows us to define nature inspired recombination rules. Moreover, if characteristics of a species can be related to each parameter of the processor, then the DNA strands defining those traits can be used. By using this trait based encoding and natural DNA recombination rules, evolution can be completely realistic.

The entire DNA strand for a particular processor is split into 2 parts. The ***Active Component*** participates in the recombination. This component represents the architectural details as well as the ***Instruction Set*** details. It is created in the encoding stage and expanded in the recombination stage i.e., the basic architectural details such as 32 bit instruction length is added to the DNA string in the encoding stage while finer aspects such as the usage of a ***Carry Save Adder*** (CSA) is added to the string during recombination stage. The ***Passive Component*** is formed in the mutation stage. It consists of the ***Finite State Machine*** (FSM) description, the ***Netlist*** and basic ***Placement*** details of the processor. The netlist defines the connectivity across various components of an electronic design while placement is the process of assigning exact

locations to various components in the chip's core area. The active components of various processors react with each other during recombination while the passive components are formed during mutation. They do not take part actively during recombination. But, the FSM details of the two processors are inherited by the offspring based on its instruction set. These inherited FSMs are then used as guidelines to form the actual FSM of the offspring.

Variable Length Delimiter Based Encoding (VLDBE) system is used for encoding. A delimiter is a special sequence which serves as a flag that represents the beginning of a node, a field, a number, a character, a symbol or any other data entity. The delimiter is set as ATCG** where * is a wild card that can be substituted by any of the symbols. Each substitution signifies a different delimiter. In particular, AA signifies ***No Delimiter***. This is very important to identify a data sequence which contains a substring 'ATCG'. For example if a data contains the string GAGCTATCGCGA, then this sequence would be transformed and encoded as GAGCTATCGAACGA. The 'AA' string signifies that the ATCG sequence was a data element rather than a part of a delimiter sequence. The 'AA' string is ignored when the sequence is decoded. Each symbol in a DNA sequence is known as a ***Dit***, which is an acronym for a DNA digit. Fig. 5 lists various delimiters and their associated sequences.

SEQUENCE	MEANING	SEQUENCE	MEANING
ATCGAA	NO DELIMITER	ATCGCA	CHILD LIST BEGINS
ATCGAT	NODE BEGINS	ATCGCT	NUMBER BEGINS
ATCGAC	DATA FIELD BEGINS	ATCGCC	NETLIST BEGINS
ATCGAG	ALHPASTRING BEGINS	ATCGCG	FSM DETAILS BEGIN
ATCGTA	NODE FUNC. BEGINS		
ATCGTT	NODE ID BEGINS	ATCGG*	SPECIAL ESCAPE SEQUENCE RESERVED FOR FUTURE USE
ATCGTC	NODE NAME BEGINS		
ATCGTG	PARENT LIST BEGINS		

Fig. 5. Table of Delimiters

As an example, a straight forward encoding scheme for number is discussed here. This encoding scheme for numbers involves a straight forward conversion to base 4 system.

The mapping of A, T, C and G is A→ 0 T→1 C→2 G→3. The numbers themselves are represented in the format shown in Fig. 6. The first field, the ***Number Delimiter*** is used to identify the sequence as a number bearing entity.

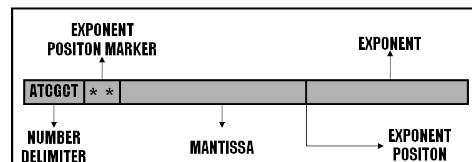

Fig. 6. Number Format

The next field is the ***Exponent Position Marker*** (EPM) which a 2 dit field. It shows where the ***Mantissa*** field ends and where the ***Exponent*** begins. The first dit of the mantissa decides its sign. If the first dit is A, then the mantissa is positive while if the first dit is G then the mantissa is negative.

If the dit is T or C, then the string has no meaning. Similarly, the first dit of the exponent decides its sign. The mantissa can have a maximum size of 16 (EPM = GG).The exponent on the other hand can be of any size. It ends when the next

delimiter is encountered. Thus numbers of any magnitude can be represented with a precision of up to16 decimal places.

5 The Gene Pools

There are two different gene pools: the ***Processor Gene Pool*** (PGP) and the ***Heuristic Gene Pool*** (HGP). The processor gene pool stores the complete design of various processors as well as several different functional units while the heuristic gene pool stores various heuristics used in the recombination and their respective effectiveness value. Architectural details, instruction set and FSM design of a microprocessor are stored as DNA sequences in the PGP. The processor strings are stored in an indexed array format for ease of retrieval. Each processor has an associated ***Potency Factor***. The potency factor is a measure of a processor's capability in producing an offspring with a specific characteristic (metrics). This is not a simple numerical value, rather it is a vector. The vector (of n metrics) stores the potency of the processor to produce various types of processors. The n metrics are decided by the user based on his opinion of a processor's goodness. For example, a user might want to use power and performance as two important metrics while another user may use chip area, which ultimately decides cost of a processor, as an important metric.

Apart from this, various components of a processor such as different types of adders, multipliers, cache structures, micro control units etc. are stored in the gene pool. These are pre-encoded by the user i.e., the exact modular structure of these units and other details are stored as DNA sequences by manually forming these strings before starting the methodology. The details that are mentioned include dependency of the module with other components, modular placement and routing details, power and performance characteristics etc. These components can be chosen from the pool based on user specification. New modules can be added to the gene pool in the specific format as and when they are created. The heuristic gene pool on the other hand stores the various heuristics employed during recombination along with their ***Degree of Belief***. The degree of belief is the amount of trust the methodology has on that particular heuristic. The heuristics are of two categories: ***Recombination heuristics*** and ***Mutation heuristics***. The recombination heuristics are operations that can be done on the two reacting sequences during recombination. These may be natural operations such as splicing or arithmetic and Boolean operations. Mutation heuristics are algorithms to optimize the given architecture and also to generate the netlist along with the FSM.

6 Recombination

The microprocessor DNA sequences are combined to form an offspring. The recombination algorithm relies on the principle of ***Localization of Reaction*** according to which the probability of a node participating in recombination is determined by its proximity to a reacting node. In simpler terms, nodes react in bunches. This nature

inspired scheme makes sense as adjacent nodes are strongly related. To achieve this, a function known as ***Spread of Recombination*** (SoR) is defined. The SoR can be any strictly decreasing function. It gives the ***Probability of Recombination*** of each node based its distance from the currently reacting node.

DNA strings can combine in many ways. In this methodology, two types of combinations are discussed. The ***Dit-Wise Recombination*** involves two strings reacting dit by dit. Moreover, heuristics mentioned in the next section are applied to a bunch of adjacent dits. There is no sense of demarcation of nodes, fields etc. This technique is biologically realistic and very random. The biggest advantage of this method is the evolution of radical offspring. On the other hand this technique may produce large number of invalid strings before providing the required architecture.

The ***Node Recombination*** involves 2 strings which react node by node. The nodes can be identified using their delimiters. Nodes adjacent to the node currently undergoing recombination also react (with a probability). Two distinct types of field recombinations are possible. ***Homogeneous Recombination*** takes places between 2 similar nodes. Since these nodes represent the same characteristic of the processor, only the values of these nodes are affected. ***Heterogeneous Recombination*** takes place between 2 dissimilar nodes. Since the nodes do not represent the same characteristic of the processor, the recombination between them results in either the loss of information of a node or the formation of a new node. In the former case, the node that is lost is reformed afterwards in the mutation stage and in the latter case the newly created node is stored as a special sequence. During the decoding phase, the user is informed about this new node. The user can then either discard this new node and revert back to the 2 nodes which combined to create them or incorporate this new node in the design of the machine.

Thus, in this manner, yet to be discovered architectural features can be evolved. For example, suppose 2 cache-less processors react with each other and the ***Latch Size*** node of the first processor combines with the ***Main Memory Size*** node of the second processor, the resultant node can be interpreted as a node bearing the information about size of the cache. Thus the concept called cache is evolved from processors which did not posses them in the first place. Thus the methodology can discover new architectural paradigms through simple combinations of existing concepts. The heterogeneous recombinations occur with a slightly smaller probability when compared to homogeneous recombinations. Moreover, even in the event of heterogeneous recombination, the probability of discarding a node and reforming it at a later stage is higher than the probability of formation of a new node. The former probability is termed as ***RFactor1*** and the latter is termed as ***RFactor2***.These parameters can be modified by the user. By default, these are set to 0.33 and 0.66. ***RFactor1* x *(1 - RFactor2)*** gives the overall probability of a new node evolving. By default this value is 0.15, which means that 3 in every 20 combinations will result in a new node. Nodes formed after recombinations are immediately validated. For example, the ***Number of Cache Levels*** can only be a natural numbers.

7 Heuristics Update Strategies

The basic heuristics involved during recombination vary from natural splicing to Boolean and binary operations. These are stored in the ***Heuristic Gene Pool*** (HGP) along with their associated ***Degree of Belief*** (DoB). The heuristics are selected as per their degree of belief; higher the DoB, greater is the probability for that heuristic being used for recombination.

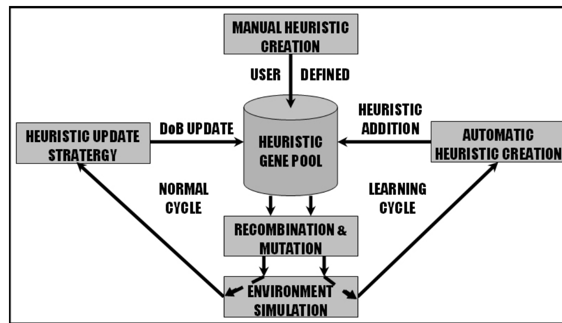

Fig. 7. Heuristic Life Cycle

Initially, all DoB are set to a default value of 0.66. The DoBs are updated once every H design cycles. Fig. 7 shows the heuristic life cycle. The basic heuristics are created beforehand. Also, user defined heuristics can be added to the HGP. Apart from these, ***Automatic Heuristic Creation*** in the ***Learning Cycle*** adds new heuristics to the HGP.

The learning cycle slightly modifies a heuristics present in the HGP and uses that for recombination. Over several iterations, various modifications of the heuristics are evaluated. If a modification yields better results consistently, then this new heuristic is added to the HGP. It is to be noted that the original heuristic is not modified; its modified version is added the HGP. In the normal cycle, each heuristic involved to evolve that offspring is noted. Once the ***Processor Evaluation*** is complete in the ***Environment Simulation*** stage, the responsible heuristics are collected in a set. Thus each iteration yields a specific set.

The ***Heuristic Update Rate*** (H) is a parameter which decides the number of iterations before which the heuristics are updated. The DoB for each heuristic is obtained. The DoB value of a set is a global DoB value assigned to all elements of the set. All DoB values are relative. The most successful processor's set gets a DoB of 1. The other sets get a relative DoB value. Next, the intersection of H sets (S_{inter}) is taken and the DoB values of each heuristic present in S_{inter} is computed by averaging the DoB values of each set in which they were originally present. At this stage various heuristics have relative DoBs ranging from 0 to 1. Next, each relative DoB is reduced by 0.5 making their range from -0.5 to 0.5. Finally, these Reduced DoBs are normalized by dividing it by the ***Normalizing Factor*** (N) which is by default set to 10. These normalized DoB values are then added to the DoB values of the respective heuristics in the HGP. Normalization is done to ensure that the DoB values do not oscillate wildly.

8 Creating DNA Mutation Algorithms

Mutation in this methodology refers to the formation of DNA strings that represent the netlist and FSM information. Once the architectural details of a processor are formed, the required functional units are fetched from the PGP. Once all the functional units are assembled, their interconnections are determined and a logical connectivity matrix is formed. Netlist generation and initial placement is carried out in the DNA domain. For this effect DNA based algorithms need to be devised. Conventional CAD algorithms for placement and routing can not be directly applied without decoding the processor string. Instead, if the DNA counter parts of these algorithms are devised, then these processes can be carried out in the DNA domain itself. Apart from perfectly suiting a DNA computational model, aspects of developmental biology can be incorporated in the interconnect development during the decoding phase. The following section describes a novel methodology for creating such DNA algorithms from conventional algorithms. Fig. 8 illustrates this methodology.

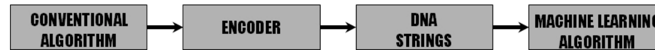

Fig. 8. Technique for Creating DNA Based Algorithms

The problem parameters at each of each step of a ***Conventional Algorithm*** is tracked and converted into a DNA string using an ***Encoder***. Thus, for every step, a set of DNA sequences are obtained. A ***Machine Learning Algorithm*** draws inferences from the changes in the problem parameters after each step and creates a corresponding DNA action step. These action steps constitute the DNA algorithm. Several examples are given to the system to enable it to learn.

9 Decoding, Optimization and Environment Simulation

The offspring DNA obtained after the mutation process contains information about the connectivity of the modules, the netlist and the FSM of the offspring processor. Though the netlist is generated during the mutation phase, its placement is not optimized completely. This phase involves decoding the offspring DNA sequence to the conventional domain and optimizing the placement of the modules and the netlist.

The offspring DNA sequence is read till a delimiter is encountered. The sequence between two node delimiters is interpreted as a node and is added to the parameter tree. The sequence inside the node is further decoded with the help of the delimiters to obtain its dependencies. In this manner the parameter tree is constructed. When an unknown node is encountered (if the node name does not match with a stored list of node names), the node is stored separately for further investigation. The details of the connectivity of the modules are also extracted from the offspring DNA sequence. The main advantage of evolution is that it is need based. The environment of a species plays a major role in the selection process and also may cause mutation of the DNA to

make the species better suited to their environment. In the context of microprocessor design automation, environment simulation involves simulating the working environment of the microprocessor. The working environment of a processor is defined as the set of applications that are to be run on it. A generalized benchmark, for both ASIC as well as General Purpose Processor, is under development.

The algorithms that are to be executed on the processor must be converted to the processor's instruction set. A ***Generic Compiler*** that takes in the processor's instruction set as an input and translates the ***Benchmarking Algorithms*** in terms of the processor's instructions is designed for this purpose. Based on the processor's architecture, the delay and the power consumed by each instruction are computed. ***Processor Simulator*** simulates the working of the processor by simulating the execution time and the power consumed by each instruction. The benchmark algorithms are virtually executed on the processor. Based on certain pre-defined heuristics, the processor characteristics are slightly varied to suit the algorithm's need. A simple example would be increasing the cache size if the algorithm involves large number of repeating instructions.

10 Simulations and Results

To test the methodology, a simple simulation was carried out. For this purpose 38 characteristics of 3 Intel processors (8085, Pentium, Pentium 4) were collected. Then their respective parameter trees were built and their DNA sequences were created using the VLDBE system. Then 8085 and Pentium 4 were recombined using the node recombination heuristics. The offsprings produced were validated. The parameter tree of valid offsprings were then correlated with the parameter tree of Pentium. The best correlation obtained every 5 trails was plotted. The maximum correlation was found to 0.97 (37 out of 38 parameters had the same value). These results are shown in Fig. 9. The simulation showed that some of the offspring of 8085 and Pentium 4 was similar to the Pentium processor.

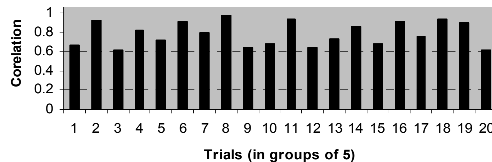

Fig. 9. The Correlation Graph

The significance of this result is that it proves that it is possible to evolve the architectural design of a microprocessor using the methodology.

11 Generalization to Other Domains

Based on the microprocessor design methodology, a general scheme to use DNA based evolution for solving design automation problems is evolved. In ***The Problem***

Domain, the problem's data is represented as such. For example, consider the design automation of an aircraft. The various parameters of an aircraft can be represented by the tree structure. These would include wingspan, no. of tail rudders, service ceiling, wing loading etc. Relationships between these parameters are found and suitable node functions are created. ***Arbitrary/ Existing Solution Creation*** would involve creating the tree for existing aircrafts. The data structure can then be encoded into DNA sequences in a biologically realistic way. After ***Encoding***, the solutions are now present in the ***DNA Domain*** where they undergo biologically realistic recombination and mutation. These reactions can be stochastic in nature guided by a few domain specific heuristics. For example the output of the homogenous combination of 2 wing structures will depend on the type of fuselage. ***Mutation*** heuristics can be learnt and these, along with recombination heuristics, can be refined using heuristic update strategies. DNA based ***Post-Processing*** is carried out to rectify minor flaws in what is otherwise a good solution. For example, if a cargo aircraft is evolved with 4 wheel landing gear, then the number of wheels can be increased based on its capacity. Solution evaluation involves environment (physical) simulations. The metrics for aircraft design evaluation could be Rate of Climb, Thrust to Weight Ratio etc. Post processing might employ conventional optimization techniques like simulated annealing, game theory, genetic algorithms etc. For example, simulated annealing can be employed to optimize the wing area. Good solutions are selectively added the ***Gene Pool***. These genes are used during the recombination phase. Moreover, the heuristics employed to obtain the current solution is also found out. The values for the goodness of the heuristics are then updated in the ***Heuristic Gene Pool***. Over several iterations, the gene pool is enriched. Both the heuristics used as well as the quality of solutions obtained is improved. After many such iterations, the final desired solution can be obtained.

12 Conclusion

This paper presented a novel methodology for naturally evolving microprocessors as per user specifications. This is only the beginning of a new vista in design automation. The encoding process can be made more biologically realistic using trait mapping techniques or amino acid based encoding schemes. Control logic synthesis can incorporated as a part of the DNA domain design automation process. Once the DNA based design automation for microprocessor stabilizes, the turn-around time would reduce drastically. Further, this methodology can be extended to other domains such as aircraft and automobile design.

References

[1] N Venkateswaran et al: Microprocessor Design Automation: A DNA Based Evolutionary Approach. BICS 2006.

[2] David B. Fogel: Evolutionary Computation: The Fossil Record. Wiley-IEEE Press (May 1, 1998)