On Evolvable Hardware

Timothy G. W. Gordon and Peter J. Bentley

University College London, U.K.

Keywords: Evolvable hardware, evolutionary electronics, evolutionary computation, genetic algorithm, automatic design, innovation, evolvability, generalization, intrinsic circuit evolution, FPGAs.

1 Introduction

Electronic circuit production is a significant industry. Ever more complex behaviors are being demanded from electronic circuits, fuelled by relentless improvements in circuit embodiment technologies. Consequently a bottleneck is developing at the point of circuit design. Traditional circuit design methodologies rely on rules that have been developed over many decades. However the need for human input to the increasingly complex design process means that modern circuit production takes one of two paths. The first is to employ more designers with greater expertise. This is expensive. The second is to simplify circuit design by imposing greater and greater abstraction to the design space. An example of this is the use of hardware description languages. This results in mounting waste of potential circuit behavior.

Recently a new field applying evolutionary techniques to hardware design and synthesis has emerged. These techniques give us a third option - to use evolution to design the circuits for us. This field has been coined *evolutionary electronics*, *hardware evolution* and *evolvable hardware* amongst others. Here it will be referred to as evolvable hardware

The field of evolvable hardware draws inspiration from a range of other fields. The most important are shown in Fig. 1. For many years computer scientists have used ideas from biology to develop algorithms for soft computing. We have seen the advent of the artificial neural network (ANN) [73] and more recently the development of evolutionary computation, the field of problem solving using algorithms inspired by evolution [10]. Ideas from nature have also been used in electronic engineering for many years. An example of this is simulated annealing which is used in many partitioning algorithms. (This algorithm is based on the physical phenomenon of annealing in cooling metals.) Recently the field of bio-inspired hardware has also developed, using ideas from biology to explore methods of fault tolerance and reconfigurability in modern hardware designs.

There is much interchange of ideas between the fields of evolvable hardware and bio-inspired hardware, but in this work we focus on the field of evolvable hardware, which lies at the crossroads between all three of these major sciences.

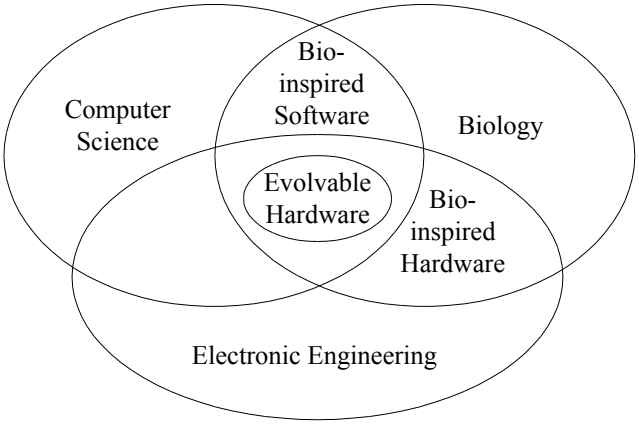

Fig. 1: The field of evolvable hardware originates from the intersection of three sciences

The interrelationships between areas of hardware design and synthesis, and evolutionary computation are shown below in Fig. 2. Digital hardware synthesis is traditionally a combination of two processes. First a human-designed circuit specification is mapped to a logical representation through the process of logic synthesis. This is represented as the lower left-hand set in Fig. 2. This netlist is then undergoes further combinatorially complex optimization processes in order to place and route the circuit to the target technology. This area is represented as the lower left-hand set in Fig. 2. Many modern EDA[1] tools use intelligent techniques in these optimization algorithms, and research into use of evolution for these purposes abounds [54, 14]. Hence we see a set representing evolutionary design optimization intersect with that of technology mapping, placement and routing in Fig. 2 to yield evolutionary mapping, placement and routing. However circuit design, along with some optimization decisions during the synthesis process are still in the domain of the human designer. It is only recently that significant interest has developed in implementing evolutionary techniques higher up the VLSI design flow at circuit design, a move that can allow evolution to generate creative designs that can rival or improve on human ones. The most widespread examples of this have been to use evolution for the design of logic, as represented by the intersection of the areas of creative evolutionary design and logic synthesis in Fig. 2. Some of the work in this intersection falls into the field of evolvable hardware. However much work at the logical level is carried out in the spirit of

[1] Electronic Design Automation

evolving programs or other forms of logic, and so is not specifically considered here.

The field of evolvable hardware is still in its infancy, and there are many problems that must be tackled before we will see large-scale industrial use of the techniques. Hence there is much active research associated with improving the algorithms used to evolve hardware. Following a brief discussion of applications of evolvable hardware in section 2, this research will be reviewed in section 3, using level of abstraction, learning bias and evolving platform as the main features to map out the area. The areas of research under review are innovation, generalization, and evolvability.

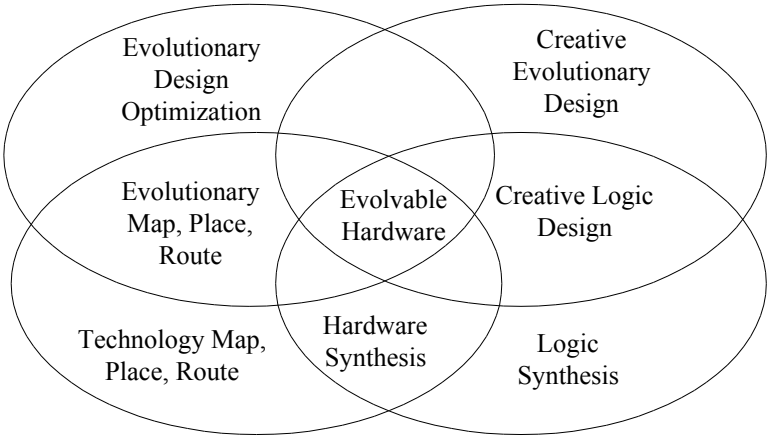

Fig. 2: Evolvable hardware can include aspects of design, optimization and traditional hardware development techniques

A particularly exciting branch of evolvable hardware research is that of *intrinsic evolution*, where evolved circuit designs are synthesized and evaluated directly on programmable logic devices (PLDs). Here the success of evolution can depend on the choice of platform. Platforms that have been used for evolvable hardware will be appraised at the end of section 3, including those that are commercially available and those developed as research tools. Two case studies will then presented in section 4. One of these is carried out in simulation and one on a commercially available platform. Finally, a summary will be given in section 5.

2 Applications of Evolvable hardware

Evolutionary Computation is the field of solving problems using learning algorithms inspired by biological evolution. These algorithms are collectively

known as *evolutionary algorithms*, and model the cycle of selection, recombination and reproduction that biological organisms undergo. Typically they work on a population of prospective solutions at any one time. Each member of the population is evaluated according to a problem-specific *fitness function*, which tests how well the trial solution performs the required task. A *selection* operator then probabilistically chooses the solutions within the population that the algorithm will subsequently focus on, based on the fitness function evaluation. The selected solutions are recombined and mutated in order to search new but related areas of the problem space. Finally they are reinserted into the population, and the process iterates. The evolutionary algorithm most commonly used to evolve hardware designs is the *genetic algorithm* [15], where each trial circuit design is encoded as a bitstring. Recombination, or *crossover*, is achieved by the probabilistic exchange of bits between individuals, and mutation by the probabilistic toggling of bits in each individual, normally according to predefined rates. Thus the algorithm explicitly separates the genetic information that is recombined and mutated (the *genotype*) from the actual circuit that is evaluated (the *phenotype*). Another evolutionary algorithm commonly used is *genetic programming* [3], where an individual solution is a computer program typically represented by a tree, with no explicit mapping between genotype and phenotype. Here crossover and mutation typically operate on branches of the trees. Variations of these and other evolutionary algorithms have been also used to evolve hardware, but for the purposes of this paper the differences between any of these are not important.

Using evolution to design circuits brings a number of important benefits to electronics, allowing design automation and innovation for an increasing range of applications. Some of the more important areas where evolvable hardware can be applied include:

- Automatic design of low cost hardware;
- Coping with poorly specified problems;
- Creation of adaptive systems;
- Creation of fault tolerant systems and
- Innovation in poorly understood design spaces.

The remainder of this section will explore these benefits in a little more detail.

2.1 Automatic Design of Low Cost Hardware

Automation of circuit synthesis has been with us for many years. Traditional digital design involves the mapping of an abstract human-designed circuit to a specific technology through the application of simple minimization, placement and routing rules. As our capability for synthesizing more complex circuits has grown, so has the need for more resourceful processes to handle the combinatorially complex mapping procedures. Intelligent techniques such as

simulated annealing [76] and ANNs [96] have been routinely used to search these exploding spaces of mappings and minimizations for some time. More recently so has evolution [54, 14].

Useful though this work on searching mapping spaces is, it is not the focus of our discussion here. Instead we are interested in automating generation of the input to this process - the circuit design - from the specified behaviors of the circuit. The required behavior may be presented in the form of pairs of input/output response required to carry out a specified computation, or some other representation of circuit behavior in cases where this is more convenient. The important point here is that to allow evolution to design, a circuit is best evaluated as a black box: according to what it does, not how it does it.

These ideas of design automation can be of significant benefit to hardware that requires a low cost per unit. One example of this is low volume hardware. Low cost reconfigurable hardware can be used to embody evolved designs. For low volume designs this reduces cost by avoiding the need for a VLSI fabrication process. Use of reconfigurable hardware also allows changes in specification to be applied not only to new applications of a design, but also to hardware already in use, thus avoiding replacement costs. Risk, and its associated cost may also be reduced, as design faults could be corrected, either by hand or through further evolution.

Evolutionary automation can even make realistic the prospect of evolving hardware designs to suit an individual. Many medical applications have not been suitable for hardware solutions owing to the expense of personalization. Evolvable hardware allows cheap fast solutions to such medical applications. For example, a system has been developed to control a prosthetic hand by recognizing patterns of myoelectric signals in a user's arm [35]. The implementation was an entirely hardware based solution with reconfigurable logic, a hardware genetic algorithm unit, a CPU core for evaluation, a chromosome memory and a random number generator implemented on the same integrated chip.

A related application is the use of evolution to tune reconfigurable analog circuits. Often the components in analog circuits differ slightly from their original specification due to variations during fabrication. It has been shown that such discrepancies can be corrected using evolution to guide the circuit to its desired behavior [66]. In this case the solution was developed for tuning intermediate frequency filters for mobile telephones to allow the use of cheaper components, and increase yields.

2.2 Poorly Specified Problems

It is difficult to specify the functionality of some problems. In these cases design automation may allow solutions to be generated from a behavioral description of

the problem. Evolution is one of a range of soft computing techniques that can be used to handle poor specifications. For instance ANNs have been applied to problems such as noisy pattern recognition for many years [73]. Evolvable hardware techniques have similarities with and advantages over ANNs, as noted by Yao and Higuchi [95]. Both can be feed-forward networks, and both can learn non-linear functions successfully. But in addition hardware is by nature a fast medium and in many cases such as when restricted to feed-forward networks, evolved hardware designs are more easily understood than ANNs. Therefore it is often suited to problems usually tackled with ANNs, but which require fast operation and good solution tractability. Evolvable hardware suitable for such purposes has already been developed for industrial use [68].

One problem where evolved hardware can rival ANNs is pattern recognition. For example high-speed robust classifiers have been developed by Higuchi et al. [24, 32]. One of the advantages of evolutionary systems is the ease with which learning biases can be incorporated. Here, robust generalization characteristics were incorporated into the solutions by specification of a bias towards short description lengths, a recommendation that results from the application of the Minimum Description Length principle to hypothesis representation in a Bayesian context [64, Chapter 6].

The ability to generate solutions to poorly specified problems can be considered as a form of creativity [72]. Creativity and innovation are important features of evolutionary processes, and will be discussed later.

2.3 Adaptive systems

With sufficient automation (i.e. real-time synthesis provided by PLDs), evolvable hardware has the potential to adapt autonomously to changes in its environment. This can be very useful for situations where real-time control over systems by humans is not possible, such as on deep space missions. It could be particularly useful when harsh or unexpected conditions are encountered.

Stoica et. al have noted that current lack of validation for on-line evolutionary systems mean that critical spacecraft control systems cannot realistically be evolved on-line [77]. This follows for other mission-critical problem control systems. However, sensors and sensory information control systems are not so critical, and their work has focused on these. Much of the data that spacecraft sensors acquire, process and transmit is highly specialized, but perhaps unknown until it is encountered. Hence systems designed to capture, process and transmit such data are suitable for evolutionary design. For instance an example of evolved hardware compression for space images is presented in [12]. Here a function is regressed for each image that is to be transmitted using genetic programming.

Other adaptive hardware compression systems have also been developed. Two

systems have been developed at the Electrotechnical Lab. (ETL) in Japan, both using predictive coding. The first breaks images into smaller sections and uses evolution to model a function for each section [74]. They also suggested that a similar system could be used for real-time adaptive compression of video signals. The other approach at ETL also aims to predict each pixel from a subset of surrounding pixels. The particular pixels that are chosen for the job are collectively known as the pixel template. This time a fixed function is used for prediction, but for each new image or image fragment, a genetic algorithm searches for the overall best pixel template to input to the prediction function. This works particularly well when the type of image to be compressed does not exhibit close correlation between neighboring pixels, but between wider ranging pixels. Hence this technique was used in [75] to compress bi-level images for high precision electrophotographic printers, images that exhibit this quality. The images were broken into smaller sections, and pixel templates to be used by a standard encoding engine (a QM-Coder) were successfully evolved. These templates and the compressed images were transmitted for decoding. It was found that this system could outperform JBIG, the ISO standard for bi-level image compression, by an average of around 50%.

Adaptive control systems have also been developed with evolvable hardware. Most commonly, this has been for robot control. For example, see [39, 70]. Industrial real-time control applications may also be suitable for this approach.

Damiani et al. have developed an on-line adaptive hashing system [8]. They have proposed that this could be used to map cache blocks to cache tags dependent on the access patterns of the data over time.

2.4 Fault Tolerant Systems

Another practical class of adaptive system is one that can adapt to faults in its own hardware, thereby implementing a level of fault-tolerance. Higuchi et al. [24] developed an adaptive hardware system that learned the behavior of an expert robot controller by example using a genetic algorithm. It could then be used as a backup controller if the expert controller failed.

On-line autonomous hardware fault detection and repair mechanisms have been developed [13, 69]. But although these architectures are examples of bio-inspired hardware and have been proposed as a platform for evolutionary experiments, they do not use evolution as an adaptive repair mechanism. Off-line systems can also be evolved to provide fault tolerance, as first shown by Thompson [81]. Thompson also showed that evolution may also generate fault tolerant solutions implicitly through the incremental nature of the evolutionary design process. Fault tolerance can exhibit itself at a population level as well as at an individual level. This is discussed in section 3.

2.5 Design Innovation in Poorly Understood Design Spaces

The design space of all circuits contains an infinitely large number of components that can be wired together in an infinite number of ways. In order to find useful circuits, human designers need to reduce this search space to a manageable size. To do this, they work in a space of lower dimensionality in which they are skilled in searching. For instance, some designers treat all components as perfect digital gates, when the components used in the embodied design are in fact high gain analog devices. The evolutionary approach may allow us to search spaces with a lower (or different) abstraction. This means that exploration of designs from the much larger, and often richer, solution space beyond the realms of the traditional hardware search spaces is possible, resulting in novel designs.

Such innovative solutions are needed when we do not have a good understanding of the design space. For instance, when compared to the logic design space, analog design is less well understood with no formal methods of abstracting the design process. Hence circuit design in this domain requires more expert knowledge. Much work has gone into using evolutionary algorithms to produce human-competitive (and better) analog circuit designs [41].

A good deal of this work centers around the optimization of parameters for current designs, and as such fits more with the field of evolutionary optimization, rather than evolutionary circuit design [2, 66]. Work in less abstract search spaces more relevant to this discussion was also been carried out. Grimbleby developed a hybrid genetic algorithm / numerical search method, using the genetic algorithm to search netlist topologies, and a numerical design optimization method to select parameter values for the evolved topologies [18]. Koza et al. and Lohn & Columbano have both evolved circuit designs at the analog netlist level [41, 49]. Handling the increase in search space when moving from optimization to design may require additional techniques which will be discussed later.

Developments in electronic engineering are beginning to generate new kinds of circuit. The design spaces of new technologies such as these are often very poorly understood. In these cases evolution can prove a useful technique in searching for innovative designs. An example of this is the field of nanoelectonics, where Thompson and Wasshuber have successfully evolved innovative (but at this stage not particularly useful) single electron NOR gates [87].

Although the digital search space is much better mapped than the spaces mentioned above, traditional logic synthesis techniques such as Karnaugh maps and the Quine McCluskey procedure are best suited to generating sum-of-products solutions. If there is a requirement to design a different representation of the circuit, for instance if it is required to optimize the map to a technology including XOR gate or multiplexers in addition to designing the logic, then the evolutionary approach can work with a design abstraction more related to the technology and potentially search areas of space that a human designer would miss if using the

techniques above. This possibility has been demonstrated in work where evolutionary algorithms have been used to discover more parsimonious circuits for representations, for instance those with multiplexer and XOR gate primitives [56].

3 Research in Evolvable Hardware

Having discussed the benefits of evolvable hardware, and some of the applications that these benefits allow, this section reviews the main thrusts of research in this field. The research is decomposed into three areas - innovation, generalization and evolvability. Before we review this work, however, it is instructional to classify the research according to a number of other features.

3.1 Classifying Evolvable hardware

A number of schemes have been developed for classifying work on evolvable hardware [97, 25, 1, 89]. We focus on three features that have been used by some of these classifications - level of abstraction, bias implementation and hardware evaluation process. Classification by level of abstraction was introduced by Hirst [25], and has been used by both Andersen [1] and Torresen [89] in different guises, using different levels of abstraction. Here we combine the levels of abstraction used by these three reviews to produce a more comprehensive means of highlighting the scale of behaviors open to the evolutionary process. We introduce a second feature - the bias implementation. With this addition we can see not only what abstractions researchers have found useful, but also how these abstractions can be imposed on an evolving system through the learning algorithm bias. This reveals a developing trend of movement from the imposition of static biases towards ones that alter throughout an evolutionary run. Finally, hardware evaluation process was a feature used by Hirst, Andersen, Torresen and Zebulum [97]. Although this classification is usually tightly linked to the level of abstraction, it will become clear that the evaluation process can lead to some important implementation choices and so is worth discussing in its own right.

Level of Abstraction

As we have already mentioned, the level of abstraction employed by evolvable hardware systems is often important. Hirst identified a number of stages within the design and synthesis lifecycle of reconfigurable hardware where the problem representation could be used as a genotype for an evolutionary algorithm [25]. This has been used as the basis for the classification of design abstraction used here, although categories that have not been applied to date have not been included, and additional levels have been added where other work in the field requires it. A diagram of the levels used here is shown in Fig. 3.

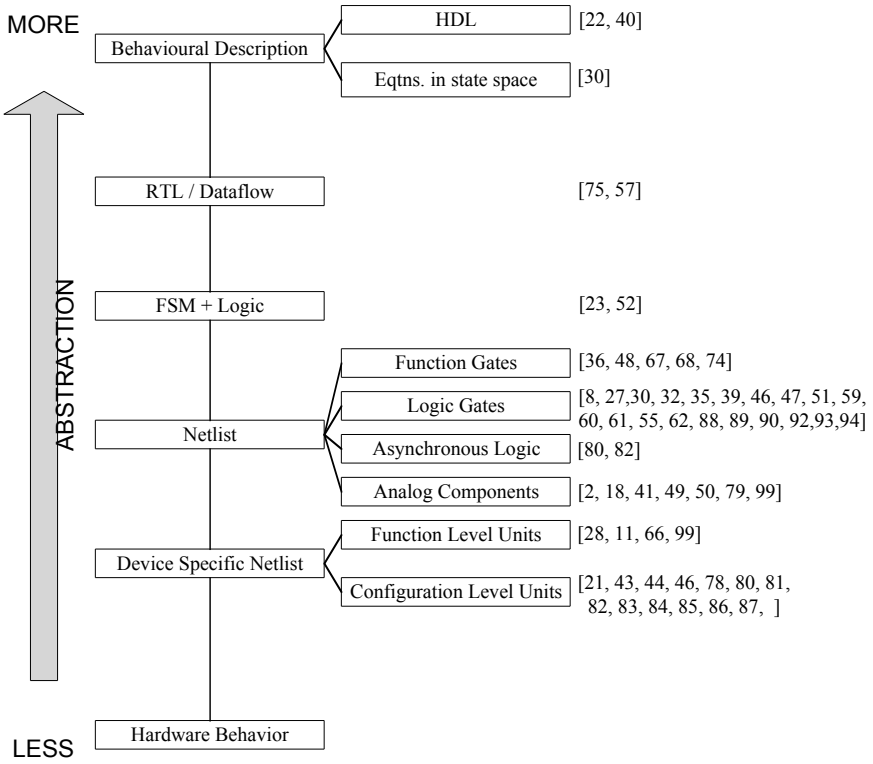

Fig. 3: Levels of abstraction needed to characterize behavior of evolved circuits.

Torresen [89] partitioned levels of abstraction into the digital and analog paradigms that are commonly used in the evolvable hardware literature, and are included here. Anderson [1] further divided the netlist level into netlists of function-level units and netlists of logical gates. We also make this differentiation. Note that the resultant levels are arbitrary levels that happen to fit most of the work in the field. Many variations of the abstractions shown here could and have been used to describe the field. However in the context of our discussion, grouping work into one of these levels allows the reader to understand the essence of work at different levels. The levels used throughout section three are displayed in Fig. 3 along with references to work that can be envisaged as evolving at each particular abstraction.

Bias Implementation

Implementation of a design abstraction is not only brought about through genotype constraint. All learning algorithms can be characterized in terms of a bias that guides the system though a space of possible solutions. These biases can be defined according to a concept learning framework introduced by Rendell [71]

and clarified by Gordon and des Jardins [17], which is illustrated in Fig. 4.

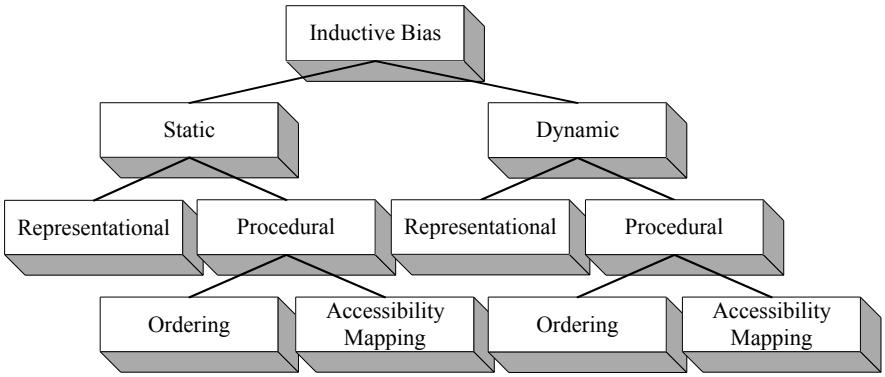

Fig. 4: Learning Biases

In this framework, the bias of a learning algorithm is defined in terms of functions operating on a solution space. Bias is separated into representational bias and procedural bias. The representational bias is the language that defines the search space within the entire space of possible solutions. We term constraints applied to the system by using such a bias as strong biases, because these biases are always obeyed. An example of this is the bias imposed by the genotype representation in a standard genetic algorithm. For the same algorithm we can make a further distinction between the representational bias that limits the search space and the representational bias that maps the search space to the solution space.

The procedural bias determines how the algorithm moves through the search space. The canonical genetic algorithm has three operators – selection, crossover and mutation, each of which exerts a different procedural bias on the search. Usually procedural biases consist of two components. The first is an accessibility mapping, which is used to map one point in search space to another. Selection and crossover have complex dynamic procedural accessibility mappings dependent on the genetic information present in the population at a given time. Mutation exerts no accessibility mapping bias, as any point in the population can be reached from any other. The second type of procedural bias accounts for mutation's effect. This is a partial ordering which is used to structure the candidate solutions in such a way that an order for search space traversal is defined. For mutation the ordering bias is that states close in space to the current state are preferred, the extent of which is determined by the mutation rate. Crossover points are chosen at random, so crossover has no ordering bias. Selection has an ordering bias determined by the fitness score awarded to each candidate solution by the evaluation function.

Dynamic representational biases can allow us to search a space of strong biases, and can easily be incorporated into evolutionary algorithms. For instance the Messy Genetic Algorithm implements such a search through new operators [16]. Another approach is that of developmental genotype-phenotype mappings [4].

Hardware Evaluation Process

Fitness values for the evaluation step of the evolutionary algorithm must be calculated for each member of the evolving population. To do this, early evolvable hardware experiments used simulations of the hardware that each member of the population specified. The main reason for this was because the production of hardware was impossible within the time-scale needed to use it for the evaluation stage of an evolutionary algorithm. Hardware designs evolved using simulation for evaluation was labelled *extrinsic* by de Garis [9], who also noted that developments in reconfigurable hardware technology could lead to the possibility of implementing solutions fast enough to evaluate real hardware within an evolutionary algorithm framework. This he called *intrinsic* evolvable hardware. Such reconfigurable hardware has been available for some time, most commonly in the form of field programmable gate arrays (FPGAs) although other devices are available. It is important to realize that the choice of an intrinsic or extrinsic system rarely determines the type of the circuit that can be evolved - it is the total design abstraction that imposes this limitation. As any abstraction can be modelled by simulation on any platform, it is possible to represent any kind of circuit if enough time and care are taken. However this may not be a feasible or sensible approach in terms of time and resources. Hence the intrinsic / extrinsic distinction is often useful as it tends to be well correlated with the use of high / low levels of abstraction.

With the three classification features of level of abstraction, bias implementation and hardware evaluation process in mind, we will now continue to review research in the field of evolvable hardware in three major threads - innovation, generalization and evolvability.

3.2 Innovation

The set of all possible circuits is a large space indeed. When humans come to design circuits, they do not consider this space. Rather, they simplify it so that the space of designs that they have to consider is much smaller. They do this by applying constraints to the search space, commonly termed design rules. This process is called abstraction.

Two common abstractions for circuits are analog and digital. Analog circuit design usually requires highly skilled practitioners who have developed sets of rules of such complexity that many see the field as an art. The need for the designer to understand the interactions within these designs eventually limit circuit complexity, as with increasing circuit complexity the interactions they must consider to search their design space effectively rapidly become unmanageable.

Designers of synchronous digital circuits use a much more constrained set of rules, which ideally reduces them to considering a sequence of circuits which can

be fully specified logically. Successive applications of design rules can eventually reduce the search space to a single circuit configuration, or a very small choice of circuits. Because of the more restrictive abstraction of digital over analog behavior, humans can discover circuits of greater computational complexity within a digital search space. This however comes at a cost - for instance digital circuits tend to require more components than their analog counterparts to achieve a computation. As a result of such trade-offs, design abstractions tend to be chosen to suit the application, and possibly the designers available.

Evolution searches large spaces objectively - given a method of evaluating design performance, it can use this to guide its search of the space. Because evolution searches the design space in a different way to humans, the abstractions that human designers rely on are no longer necessary, and in many only serve to restrict evolution from finding novel designs.

3.2.1 Relaxation of Abstractions

Seminal work in this idea was carried out by Thompson. He first set out to show that evolution could successfully manipulate the dynamics and structure of circuits when the dynamical and structural constraints that human designers depend on heavily on had been relaxed. In [84] he demonstrated this by evolving a complex recurrent network of high-speed gates at a netlist level abstraction to behave as a low frequency oscillator. Fitness was measured as an average error based on the sum of the differences between desired and measured transition periods. Circuits were evaluated in simulation using an asynchronous digital abstraction. If we interpret this in terms of bias, we see a static representational bias was used to ensure only the space of recurrent netlists of logic gates was searched. The simulator imposed a static procedural ordering bias on the selection operator through the evaluation function. Hence the space of any behavior of electronic circuits not modelled by the simulator was not searched. On the other hand, the selection operator could explore the asynchronous dynamics afforded by the simulation, free to make use of any such behavior or ignore it as it saw fit.

The required behavior of the circuit was successfully evolved, showing that it is possible for evolution to search without the constraints usually needed by human designers. Further, a graph-partitioning algorithm showed the structure of the circuit contained no significant structural modules as would be seen through the successive abstraction approach of a top-down human approach. In addition circuit behavior relied on methods that would not have been used by human designers. So not only had evolution found a solution by searching the space beyond conventional circuit design space, but also it had found one that that actually lay in this space.

Thompson went on to show evolution with relaxed restrictions on circuit dynamics was possible in real hardware. The hardware was a finite state machine for a robot controller. However whether the states were controlled synchronously

by a given clock or not was under genetic control. The evolved robot controller used a mixture of synchronous and asynchronous behavior, and interacted with the environment in a complex dynamical manner to produce behavior that would not have been possible using the finite state machine abstraction with such limited resources. Again evolution had found a solution through relaxation of a human design constraint, in this case the constraint being the synchrony imposed on the finite state machine abstraction. But in addition evolution has found a circuit that uses the rich dynamics that can arise by relaxing design constraints, demonstrating that such dynamics technique can be useful.

Thompson also carried out the first intrinsic evolution of a circuit evaluated on an FPGA. A 10x10 area of a Xilinx 6126 bitstream was evolved. Almost all bits in the bitstream corresponding to this area were evolved, directly as the bits of the chromosome of a genetic algorithm [80]. Thereby Thompson set about evolving a circuit at the lowest level of abstraction possible - that of the physical behavior of the target technology. The task was to evolve a circuit do discriminate between 1kHz and 10kHz signals. Fitness was calculated by subjecting each circuit to five 500ms bursts of each signal in a random order, and awarding high fitness to circuits with a large difference between the average voltage of the output during these bursts. The average voltages were measured with an analog integrator. The only input to the circuit was the 1kHz / 10 kHz signal - no clock was given. Again, a highly innovative circuit was evolved that used a fraction of the resources that a human designer would need. Following months of analysis Thompson and Layzell described the functionality of the circuit as 'bizarre' and to date the nature of some of the mechanisms it uses are not completely understood [85].

Thompson and Layzell carried out a similar experiment, this time providing the circuit with a 6 MHz oscillator signal, which could be used or ignored as evolution required [86]. The prime motivation for the experiment was to investigate robustness, and so evaluation was carried out under a range of conditions specified by the operational envelope detailed in section 3.3. Hence the constraints to the system were the same as before except that a weak bias towards robust behavior had been added. However an additional dynamical resource had been provided. The resulting circuit made use of the clock, and the design was simulated using the PSpice digital simulator. The simulated design behaved exactly as that of the real circuit, showing that evolution had found a solution within the digital design abstraction, even through the constraints did not require that. However, analysis of the simulation waveforms showed a large number of transient signals. This led Thompson to hypothesize that evolution can find circuits within the digital abstraction, with all the benefits of robustness and technology insensitivity that this entails, but that these circuits may lie outside the scope of the human digital design space. To put it another way, evolution can find innovative digital designs by searching a bigger space than human designers, but by using a different mapping though space.

3.2.2 Combining Common Abstractions

Miller et al. have also conducted research into the discovery of innovative circuits, one of their main motivations being the derivation of new design principles. In [55] they note that Boolean or other algebraic rules can map from a truth table of required circuit behavior to an expression in terms of that algebra. They then suggest that a bottom-up evolutionary approach can search not just the class of expressions that the algebraic rules map to, but a larger space of logical representations, beyond commonly used algebras.

In an aim to demonstrate this they successfully evolved one and two bit adders based on the ripple adder principle using a feed-forward netlist representation of AND, OR, NOT, XOR and MUX gates. This space lies beyond the commonly used Boolean and Reed-Muller algebra spaces, but is of interest as the multiplexer is available as a basic unit in many technologies. Hence this approach is very similar to Thompson's approach in principle - that the discovery of innovative circuits can be facilitated through the modification of design abstractions implemented through representational biases.

Many of the circuits reported in this and other work [56, 62] were unusual but interesting because of their efficiency in terms of gate count. They lay in the space of circuits making use of multiplexers and XOR gates, outside the space of traditional atomic Boolean logic units. They argued that these circuits were unlikely to be found using traditional algebraic methods, and so evolutionary "assemble-and-test" is a useful way that such a space can be explored. The work continued with the evolution of two bit and three bit multipliers. All work was carried out in simulation. Similar work has been carried out with multiple valued algebras [36].

Another aspect of this group's work is the contention that design principles could be discovered by searching for patterns in evolved circuits. In particular they hypothesis that by evolving a series of modules of increasing size, design principles that they have in common may be extracted from them. In [55] and [62] they evolved many one and two bit adders, and by inspection deduced the principle of the ripple adder. Although this principle is known, they went on to argue that evolution discovered and made use of it with no prior knowledge or explicit bias. As the design principle could be extracted from comparing one and two bit adders that had evolved to use the principle, they asserted that evolution could be used as a method of design principle discovery.

Their recent work in this area has concentrated on developing an automatic method of principle detection [57]. Having successfully evolved two and three bit multipliers that are much more compact than those of traditional design, they have integrated a data mining procedure [33] to search for design principles. The learning algorithm used for the data mining process is an instance based learning technique called Case Based Reasoning [64, Chapter 8].

3.3 Generalization

Inductive learners such as evolutionary algorithms infer hypotheses from observed training examples. If it is infeasible for all possible training examples to be observed by the learner, it must use an inductive bias to generalize beyond the cases it has observed. This is a very common situation. For example, a combinational circuit with eighty input pins has 2^{80} possible sets of inputs. Even if we observe each of these training examples at the rate of one million examples a second, observing all the examples only once would take over three times the age of the universe. Choosing a good inductive bias is therefore crucial to the success of our learner.

Two approaches to generalization can be found in the evolvable hardware literature:

(a) Introduce knowledge about the *nature* of circuits that have the generalization characteristics required, perhaps in the form of a heuristic.

(b) Introduce knowledge about the *behavior* of circuits that have the generalization characteristics required, and make evolution learn about the nature of such circuits in addition to the primary task.

Both of these can be applied either as a strong bias in which case a circuit with the generalization capability follows deductively, or a weak bias in which case the other biases innate in the evolutionary process must make up the shortfall in knowledge.

3.3.1 Generalization Across Inputs

Iwata et al. have successfully managed to improve the generalization abilities of evolved pattern recognizers in the manner of case (a) above [32]. They introduced a heuristic commonly used in the machine learning literature to improve generalization. The heuristic results from the application of the Minimum Description Length (MDL) principle to the discovery of maximum a posteriori hypotheses in Bayesian settings, and biases the search towards small circuits. For details of this interpretation of MDL see [64, chapter 6]. Miller and Thomson investigated the generalization abilities of a system evolving a three bit multiplier with respect to the size of its input training set [60]. The task was to evolve a functional circuit from a subset of the truth table. They found that if evolution was presented with a subset of training cases throughout the entire evolutionary run it was not able to produce general solutions. This suggests that in the setting of this problem and algorithm there was no implicit bias towards generality. They also reported that even when evolution was provided with a new set of training cases randomly drawn from the truth table every generation, general solutions were still not found. This is an example of case (b) above, and would suggest that evolution had little memory in the context of this problem.

Miller and Thomson also investigated the evolution of square root functions [61]. In these cases, they discovered that some acceptable solutions were generated when evolution was limited to an incomplete training set. These cases occurred when the missing training cases tested low-order bits, which contributed less to the fitness. Hence they concluded that real-valued functions could be approximated.

Imamura, Foster and Krings have also considered the generalization problem [31], and concluded that evolving fully correct circuits to many problems was difficult. They pointed out that the problem was exacerbated in functions where each test vector contained equal amounts of information relevant to the problem, such as the case of the three bit multiplier studied by Miller and Thomson. However they suggested that in cases where the data contained a large amount of 'don't care' values evolvable hardware could be successful using a smaller test vector. They suggested suitable applications might be feature extraction, data mining and data cleaning.

3.3.2 Generalization Across Operating Environments

It is unrealistic for the algorithm to train from every conceivable circuit input. It is also unrealistic to train under every conceivable operating environment. Operating environments might include a range of technologies or platforms on which the designed circuit should operate, and a range of conditions that the embodied circuit may be subjected to.

Human designers often manage such generalization by imposing strong biases on the nature of the circuit. These biases are representational abstractions that are known to produce behavior common across all necessary operating environments. The abstractions are then mirrored on the physical hardware through some constraint on its behavior. A circuit that behaves correctly in all necessary conditions should then follow. For instance, the digital design abstraction requires that the physical gates of the target technology behave as perfect logic gates. In reality, they are physical devices that behave as high gain amplifiers. Timing constraints and operating environment constraints specified by the manufacturer of the physical device are imposed on the real hardware. This ensures that when an abstract computation takes place the voltages of the gates have saturated, any transient behavior generated before saturation has dissipated. From this point, their outputs can be treated as logical values. In synchronous systems, these constraints are usually imposed with respect to a clock. The manufacturer will then guarantee that for a range of operating conditions, the device will behave as it appeared to within the design abstraction. The design is then portable across a range of devices and operating conditions.

When automation is a more important requirement than innovation, evolutionary circuit design often takes a similar approach to the human design process. Consequently, human design abstractions are often used by evolutionary

algorithms to ensure that certain familiar behaviors are embedded in the design. This is most easily done by imposing a representational bias. For example the Electrotechnical Lab in Tsukuba has centered on a netlist level design abstraction, implemented through a static representational bias. Early experiments were designed around the AND-OR networks of a Lattice GAL16V8 PLA. Test circuits were evaluated using a logic simulator, as this device could only be reprogrammed in the order of 10^4 times before failing. A circuits behavior in simulation is its behavior in one operating environment. In order for the behavior of the simulation to generalize to any GAL16V8 operating under standard conditions, a further representational bias was imposed such that no feedback was permitted. This prevented the search including any circuits that would rely on timing-specific features of the gates in simulation, and so the behavior of the circuit in both hardware and simulation would be identical. Note that in practice a real implementation of the evolved circuits would have a physical delay before the desired output signal reached the output. This would be dependent on the delays of the actual gates on that particular chip and the operating environment. Initially a 6-mulitplexer was evolved [23].

Work with the GAL16V8 at ETL broadened to use of a finite state machine (FSM) abstraction. First the same netlist representation was used to evolve the state transition functions of a counter given the counter inputs [23]. This was extended to the successful evolution of a four state Mealy FSM with one input and one output. The FSM abstraction was applied through the fitness evaluation, a static procedural ordering bias. This is a weak bias. Consequently circuits corresponding to FSM, and thereby providing the generalized states were not the only points searched. In fact function FSMs only appeared towards the end of the evolutionary runs. It is not clear whether this additional exploration was advantageous. Similar work has been carried out by Manovit et al. [52].

3.3.3 Generalization Across Operating Environments using Behavior

In cases where no knowledge about the nature of solutions that generalize across all operating environments is available, the only solution is for evolution to infer this information from examples.

Early work with intrinsically evolved circuits by Thompson focused on design innovation through relaxation of constraints [80]. He successfully evolved an innovative circuit to distinguish between two frequencies, using a Xilinx 6200 FPGA. However he then went on to note the lack of robustness to environmental conditions such as temperature, electronic surroundings, and power supply may occur. It was also noted that the design was not portable, not only when moved to a different FPGA, but also a different area of the same FPGA. Similar results have been reported by Masner et al. [53]. We can think of Thompson's solution as including the bounds of the operating requirements as a procedural ordering bias, although not one specified directly by the fitness function [83]. He took a

previously evolved FPGA circuit that discriminated between two tones. He then specified a number of parameters for an operational envelope which when varied affected the performance of this circuit: temperature, power supply, fabrication variations, packaging, electronic surroundings, output load and circuit position on the FPGA. The final population from the previous experiment was then allowed to evolve further, this time on a five different FPGAs maintained at the limits of environmental conditions specified by the operational envelope parameters. Although there was no guarantee that the circuit would generalize to behave robustly to all environmental conditions within the envelope, Thompson found a level of robustness evolved in four out of five cases. Hence, it appears that the biases of the evolutionary algorithm and the 6200 architecture promoted good operating condition generalization characteristics. It is interesting to note that incorporating a procedural bias towards generality in this way was successful when applied to operating conditions, but not in the case of Miller's input test cases discussed in section 3.3.1.

Another example of this is the portability problem of evolving analog circuits extrinsically. Analog circuit simulators tend to simulate circuit behavior very closely. Hence we would expect extrinsically evolved circuits to generalize well to the real circuit. However this does not happen in practice. One issue is that some behaviors that simulate according to the physics programmed into the simulator may not be feasible in the chosen implementation technology. A common example is that simulators often allow the use of extremely high currents. Koza et al. have evolved many circuits extrinsically at an analog abstraction using the Berkeley SPICE simulator [41], but have not been able to build them in real life because of such behaviors. Additionally analog simulators use very precise operating conditions. The circuits of Koza et al. are evolved to operate at 27°C, and so there is no explicit bias towards generalization across a range of temperatures.

Portability both between simulated and real environments and between changing real environments is problem for robot controllers. In order to develop robustness in both cases Keymeulen et al used a combination of both generalization techniques - they introduced the robot to as many simulated environments as possible and also introduced mutation operators designed to generalize the evolved circuit model.

Stoica et al. have evolved networks of transistors intrinsically and suffered from the reverse problem - circuits evolved intrinsically operate well in hardware, but may not in software [78]. Their solution to the problem is to impose a procedural ordering bias towards working in simulation and in hardware by evaluating some circuits of each generation intrinsically, and some extrinsically. This they term mixtrinsic evolution [79]. They also suggested that another use of mixtrinsic evolution would be to reward solutions which operate differently in simulation than when instantiated in a physical circuit. This would place a procedural ordering bias towards innovative behavior not captured by simulation.

3.3.4 Inherent Generalization

One other possibility is that the biases of the evolutionary algorithm have an inherent tendency to generate solutions that generalize across certain conditions. Thereby evolved circuits would exhibit robustness to changes in those particular conditions "for free".

One example of such robustness is robustness to faults. The obvious method of evolving such robustness is to include a requirement to robustness in the fitness function, thereby altering the procedural ordering bias of the selection operator [83]. This method of generalization has already been discussed.

Thompson has also postulated that evolved circuits may be inherently robust to some types of fault. He observed that an evolutionary algorithm will by nature be drawn to optima surrounded by areas of high fitness, and suggested that as a result, a single bit mutation from such an optimum will also tend to also have a high fitness. He then conducted experiments on an artificial NK landscape to demonstrate this. For details of this type of landscape see [38]. He then proposed that such an effect could have beneficial engineering consequences if a mutation causes a change in the circuit that is similar to a fault - namely that the evolved system is likely to be inherently robust to such faults. He went on to highlight this using the evolution of a state machine for a robot controller as an example. The state machine used a RAM to hold a lookup table of state transitions. Each bit of the RAM was directly encoded in the chromosome, and so mutation of one of these bits had a similar effect to a 'single stuck at' (SSA) fault. Examination of the effect of SSA faults on a previously evolved state machine revealed that it was quite robust to faults. However as state machines for this problem with similar fitness could not be easily generated by any means other than evolution, statistical tests of the evolved machine's resilience to faults could not be carried out. Even so, the idea of a general characteristic developing inherently as a result of a mapping bias inducing the same move through space as a change in environment seems reasonable, and in some cases may be easier than specifying a generalization requirement directly through the procedural ordering bias.

Following this, Masner et al. [53] have carried out studies of the effect of representational bias on the robustness of evolved sorting networks to a range of faults. The aim of the work was to explore the relationship between size and robustness of sorting networks using two representations - tree and linear. They noted that robustness first increases and then decreases with size, and is therefore not due purely to the existence of redundant non-functional gates in the sorting networks. They also noted that the linear representation tended to decrease in robustness with respect to size faster than the tree representation. Again this demonstrates that robustness is a feature that can be evolved through the proper selection of procedural ordering and representational biases.

Layzell has suggested that robustness of solutions can also be generated at the

level of populations [44]. In particular he was interested in the ability of another member of the population to be robust with respect to a fault that causes the original best solution to fail. This he called populational fault tolerance (PFT). He went on to demonstrate that PFT is inherent in certain classes of evolved circuit, and test various hypotheses that could explain its nature [45]. As with Masner et al. he noted that fault tolerance did not seem to be a result of non-functional redundancy in the current design. Instead he showed that descendants of a previously best and inherently different design were still present in redundant genes in the members of the population. It was these individuals that resulted in PFT.

3.4 Evolvability

The evolutionary paradigm is no panacea for searching the extremely large spaces that we have been discussing. Researchers have found evolution of small low complexity circuits such as a three bit multiplier difficult [60]. For evolution to search for useful (i.e. large and/or complex) designs over large spaces, we need to make the search for incremental improvements by the evolutionary algorithm easier.

Traditional theory in evolutionary computation suggests that evolution cannot continue to search effectively after a population has converged. [15, 63] Hence from this point of view, making evolution's job as easy as possible improves the quality of the search, as well as the speed.

Intuitively, one would expect evolvability to be improved by limiting the search to a smaller space than the space of all circuits possible in a target technology. However we do not have to limit the space using human design rules - we could apply new abstractions, which assist the method of search that evolution employs. These abstractions may reduce the size of the space that evolution searches, or they may transform the space into something of the same size (or bigger) that is more tractable to evolution.

Finding useful abstractions may be very hard. However we could allow evolution itself to search for abstractions that it finds useful. Also, these abstractions no longer need to be hard constraints that must be adhered to, but may be soft constraints used to bias the search without forcing it to avoid potentially useful areas of space. (The flip-side of this is that evolution is given more work to do.)

3.4.1 Static Representational Biases

Choosing a good representational bias is very important to the success of the algorithm. As discussed in the section on innovation, we must ensure that solutions we are interested in finding can be represented. However there is another issue. Because the directions of many of the dynamic biases within a genetic algorithm depend on the space around them, the choice of representation will

affect the efficiency of the algorithm. Work has been carried out on how static representational biases can affect the evolvability of logic netlist design spaces. Miller and Thomson explored how changes in circuit geometry [60], and how functionality-to-routing ratio affected evolvability [59]. Both appeared to be important. In the case of geometry little else was concluded but for functionality/routing more could be said. They found that forcing differentiated routing and functional units for evolution was important, but functional resources were more important than routing. They also noted that the importance of tuning the average number of neighbors to each cell. Kalganova et al. have analyzed how representational biases can affect the geometry both of multiple valued logic netlists [36] and Boolean logic netlists [37]. However all these studies are likely to be dependent on the problem and the other biases employed, making it difficult to draw general conclusions.

3.4.2 Function Level Evolution

The function level approach to improving evolvability was developed at ETL, and has been adopted by many others. Early evolvable hardware work at ETL used a combinational digital netlist representation that could be easily mapped to hardware. For harder problems they suggested that the size of the chromosome limited the speed of evolution. They suggested two solutions - one was to use more abstract structures in the representation, thereby reducing the search space. This was dubbed function-level evolvable hardware [48, 67]. The difficulty here is in choosing the correct structures to use in the representation. Any abstraction made makes assumptions about the type of problem. Therefore problem-dependent functions would have to be developed for each class of problem. Once this trade-off has been made, evolution is now limited to search the space of this abstraction, and any innovative solutions at the lower abstraction will be unattainable.

In response to the work on function level evolvable hardware Thompson argued that such course-grained representations could reduce the evolvability of a hardware design space [82], noting that as the control evolution has over an evolving platform becomes coarser, so can the fitness landscape. This can result in less evolvable landscapes, at the limit reducing evolutionary search to random search. Instead, Thompson argues that traditional evolution has the capability to search larger spaces than are advocated by those at ETL in [48, 57]. In particular he suggests that there may be features of many hardware design landscapes that allow us to search large spaces beyond the point where the evolving population has converged in fitness. Such a feature, he suggested, was the neutral network.

3.4.3 Neutral Networks

If the genotype search space is considered as a high dimensional fitness landscape, neutral networks can be conceived as pathways or networks of genotypes whose

phenotypes share the same fitness. It has been suggested [29] that genetic drift along such networks can allow evolution to escape local optima they would otherwise be anchored to. The idea of neutral mutations has been recognized in the field of evolutionary biology for some time but has only recently been used as a paradigm for search in evolutionary computation. Taking advantage of such a paradigm requires the evolutionary algorithm to be modified to include this knowledge. To this end Harvey [20] developed the Species Adaptation Genetic Algorithm (SAGA), which advocates incremental changes in genotype length and a much greater mutation rate than is common for genetic algorithms. Thompson however used a fixed length genetic algorithm with a SAGA-style mutation rate to search an incredibly large circuit design space (2^{1800}) for good solutions. This he succeeded in doing, and when the algorithm was stopped owing to time constraints, fitness was still increasing even though the population had converged long before. [21] Analysis of the evolutionary process did indeed reveal that a converged population had drifted along neutral networks to more fruitful areas of the search space. He put much of this behavior down to the increased mutation rate, a change to the static procedural mapping of the algorithm. (In this paradigm, mutation usurps the role of the genetic algorithm's primary variation operator from crossover) He noted that the nature of the solution representation space was also important – without the existence of neutral networks in the solution space would not be possible, and speculated that neutral networks might be a feature of a great deal of design spaces including many hardware design spaces.

Vassiliev and Miller also endorse the neutral network theory. Their work on neutrality in the three bit multiplier logic netlist space [93] suggests that neutral changes at the start of an evolutionary run occur because of high redundancy in the genotype. As the run continues and fitness becomes higher, redundancy is reduced. However the number of neutral changes does not drop as quickly, suggesting that selection actually promotes neutral changes in order to search the design space. They then went on to show that when neutral mutations were forbidden, the evolvability of the landscape was reduced. Comparisons between the three bit multiplier space and other problem spaces with the same representational bias suggested that the nature of the spaces was similar, and so the results from this space may well hold for others. They have also proposed shown that the search for innovation may be assisted by using current designs as a starting point for evolution, and proposed that neutral bridge could be used to lead us from conventional design space to areas beyond. [92]

Much of the work on neutrality uses evolutionary strategies as opposed to the more traditional genetic algorithm. Evolutionary strategies do not use the crossover operator. Because of this their use in studies of neutral mutations, the driving force of evolution in the neutral network paradigm, simplifies analysis.

3.4.4 Dynamic Representational Biases

The second proposal from ETL to improve the speed of evolution was to use a

variable length representation. This imposes a dynamic representational bias – it allows evolution to searches the space of representations in addition to the problem space. If successful, a representation useful to the algorithm can be found. This is a good idea, as the search is be directed by the procedural ordering bias imposed by the problem, rather than a bias of the algorithm, meaning the technique could be portable to a range of problems. Applied to a pattern recognition problem, the results were greatly improved over an algorithm that did not search the bias space, both in terms of solution parsimony and efficacy [34]. However, the evaluation of the chromosome was such that the representational space was still always limited to a feed-forward network of Boolean gates.

A similar approach was taken by Zebulum in an experiment to evolve Boolean functions using a chromosome of product terms that were summed by the fitness function [98]. However the search order of representation space differed from the ETL experiments. Inspired by the observation that complex organisms have evolved from simpler ones, the population was seeded with short chromosomes. A new operator was introduced to increase chromosome length, under the control of a fixed parameter. Hence a fixed dynamic bias to move from short representations to long ones was set. It was found that a low rate of increase allowed fully functional, but more parsimonious solutions to be found over a larger rate.

An interesting approach was that of the Adaptive Architecture Methodology (AdAM) system [22]. Here a system was developed to evolve a hardware description language (HDL). These are high-level behavioral languages, but also allow register transfer level constructs to be embedded within the code. The approach taken here was to evolve a tree, each node of which referred to a rewriting rule from a set that effectively make up the grammar of the HDL. As the mapping between genotype and phenotype depends on the structure of the genotype itself, this becomes a method of searching representation space - it is a dynamic representational bias. This allowed evolution to find representations in which problem-dependent structures or modules could be better represented.

Modularity has been recognized as important in evolutionary systems for some time, particularly in genetic programming [3, 41] and more recently in evolvable hardware [57]. For this purpose, Kitano has proposed a similar rule-based system that promotes large-scale modularity [40].

The representation used by Koza has similar properties. It is a tree of circuit modifying functions that act on an embryonic circuit. It is this mapping between the embryo and the phenotype that is evolved, again allowing search of the representational bias space. The rules are chosen to avoid creation of invalid circuits [41]. Lohn and Columbano have used a similar approach, but with a linear mapping representation which is applied to an embryonic circuit in an unfolding manner, rather than a circuit modifying one. Although its representational power is limited, it has been found to be effective. [49]

3.4.5 Dynamic Procedural Biases

Rather than finding new ways in which to express the solution, we can search the space of ways to express the problem.

Torresen recognized both the benefits of modularity and the advantages of allowing evolution to search for its own modules. He has suggested using what we can interpret as dynamic procedural ordering biases to search for them. The first is to partition the training vector for a problem by hand and evolve solutions to each separate section. He applied this to a character recognition problem and gained promising results [88]. The second method he suggested was inspired by the concept of incremental learning introduced by Brooks [5]. This bias advocates presentation of training vectors representative of an increasingly large set of behaviors to a single learner over time. This way, more complex behavior can be learned over time. Both methods provide a way of generating problem modularity, but they both require intervention from the designer in order to determine how to break up the problem into sets of behaviors. Torresen showed this technique improved evolvability for a real-world image recognition problem [90].

Lohn et al. have also worked on dynamic procedural biases. They compared three dynamic fitness functions against a static one. [50] The dynamic fitness functions increased in difficulty during an evolutionary run, directly altering the procedural ordering bias. One had a fixed increase in difficulty, one had a simple adaptive increase based on the best fitness within the population, and one put the level of difficulty was under genetic control by co-evolving the problem and the solution. In the former cases the newly introduced biases are new procedural ordering biases. In the latter, a complex interaction between the procedural ordering and mapping biases of two evolving systems is set up. The results showed that static and co-evolutionary biases performed best on an amplifier design problem.

The evolvability of two bit multipliers have also been studied with respect to the evolutionary operator bias, a procedural ordering bias [94]. Uniform crossover and point mutation were studied. Analysis of the ruggedness of the landscape showed that uniform crossover generated a large amount of ruggedness and was therefore unsuitable for searching this space. Point mutation fared much better.

3.5 Platform Research

We have now reviewed most of the research into evolvable hardware. We have seen that many researchers believe that working at low levels of abstraction can have advantages. We have also seen mechanisms to deal with the increased size of the search space, if needed at all, are being actively investigated. What we have not considered is the availability of platforms for low abstraction hardware evolution.

In this section, we cover the platforms that have been reported in the evolvable hardware literature. Some are commercially available, and some have been developed by researchers. Of these, few have been developed with evolvable hardware as a primary goal, and so suffer from various shortcomings. Some have been developed expressly for evolvable hardware. However, industry is already interested in evolvable hardware [24, 66], and as research tools all of these solutions are unlikely to be available to industry for real-world experiments. Even if they are, the cost associated with low-volume research platforms may be prohibitive.

Other devices are commercial, and have not been designed with evolvable hardware in mind. Because of this, most struggle to compete with dedicated evolvable hardware on performance, versatility and ease of use for our purposes. However they have the advantages of availability and cost, and so are a more likely candidate for future industrial applications.

3.5.1 Criteria for successful evolutionary platforms

In [82] Thompson listed a number of criteria for intrinsic circuit evolution platforms. These are discussed below:

Reconfigurable an unlimited number of times - Many field programmable devices are designed to be programmed only once. Others are designed to be programmed a small number of times, but repeated configuration can eventually cause damage. Evolutionary experiments can require millions of evaluations, and so devices for intrinsic experiments should be able to be reconfigured infinitely.

Fast and / or partial reconfiguration - If millions of evaluations are needed, the evaluation process should be fast. Modern programmable devices have millions of configurable transistors and consequently have large configuration bitstreams. This can mean that downloading the configuration becomes the bottleneck of the evolutionary process. The brute force solution to this problem is to use devices with high bandwidth configuration ports. Another solution is to evaluate many individuals at once, as proposed by Higuchi amongst others. [28] Batch evaluation limits the type of evolutionary algorithm to those with large populations, ruling out the use of steady state genetic algorithms, or low-population evolutionary strategies. A more elegant solution is that of partial reconfiguration, where only the changes from the current configuration need to be uploaded. This yields similar bandwidth use with no constraints on the learning algorithm.

Indestructibility or Validity Checking - In conventional CMOS technologies, a wire driven from two sources can result a short circuit if one drives the wire to a different voltage level than another. The high currents generated from such an event are extremely undesirable, as they can damage the device, and so should be prevented by hard constraints, rather than the softer ones advocated so far. Some

hardware platforms are designed around an architecture with which contention is impossible. For those that are not, there are two options - either an abstract architecture can be imposed on top of the real hardware, or circuits can be tested for contention before they are synthesized, and evaluated by an alternative means if such a condition is detected.

Fine Grain reconfigurability - In order to allow evolution the ability to innovate, evolution must be able to manipulate candidate circuits at a low level of abstraction. Hence a good platform needs fine-grain control over the evolving platform. Thompson also points out the distinction between fine grain architectures and fine grain reconfigurability - namely that although a device's architecture may be based on repeated large units, if these can be reconfigured at a finer level then this criterion will be met.

Flexible I/O - The method of supplying input and retrieving output from an evolved circuit can affect the feasibility of successful evolution, and so a platform that allows experimentation with this is useful.

Low cost - This is of particular importance when the motive behind using evolution is to lower costs through design automation.

Observability - In order to analyze how evolved circuits work, their internal signals need to be probed. Although when working with low design abstractions it may be impossible to avert the potential of signal probes to change the behavior of the circuit, and the probed signal, architectures should be chosen with this as a consideration.

3.5.2 Platforms

Whilst bearing these criteria in mind, the platforms that have been used or proposed for use for evolvable hardware experiments are now considered briefly. These can be classified into three groups - commercial digital, commercial analog and research platforms, and are tabulated below.

Commercial Analog Platforms
Zetex TRAC [11] - Based around two pipelines of op. amps. Linear and non-linear functions successfully evolved. Large grained reconfigurability and limited topology limit worth for evolution. Has been used with evolvable motherboard to provide external components.
Motorola MPAA020 [99]- 20 cells containing an op. amp, comparator, transistors, capacitors and SRAM. Range of circuits have been evolved. Much of the bitstream is proprietary. Geared towards circuits based around the op. amp.

Commercial Digital Platforms
Xilinx 6200 [80, 86, 41] - Developed for dynamic reconfig. apps. Fast and infinite reconfig., fully or partially. Homogenous fine-grained architecture of MUXes. All configurations valid. Good I/O. Expensive, no longer produced.
Xilinx XC4000 [46] - Low cost, infinite but slow reconfig. SRAM LUT based architecture. Damaged by invalid configurations. Parts of bitstream proprietary and undisclosed. Reconfigurable at resource level using Xilinx JBits software.
Xilinx Virtex [27, 47] - Medium cost. Can be reconfigured infinitely and quickly, fully and partially. Can be damaged by invalid configurations. Some of the bitstream is proprietary and undisclosed, but most hardware resources can be reconfigured Xilinx JBits software. Widely available.

Research Platforms
Field Programmable Transistor Arrays [78, 79] - Reconfigurable at transistor level, additionally supporting capacitors and multiple I/O points. Programmable voltages control resistances of connecting switches, hence they act as additional transistors. Flexible enough to evolve both filters and amplifiers. Fits criteria for evolvable hardware well.
Field Programmable Processor Arrays [13] - Bio-inspired fault tolerant architecture. Early prototypes used a multiplexer as the reconfigurable unit. New revisions increased the granularity of the unit under genetic control through decision tree machine to RISC processor. Now better suited to evolution of more abstract structures, e.g. genetic programming.
Palmo [19] - Based around array of integrators. Signals encoded using PWM. All configurations valid. Integrator unit is useful for evolution of analog and mixed signal processing circuits. Beyond this, not so versatile.
Programmable Transistor Array [42] - 16x16 array of programmable PMOS and NMOS transistor cells. Each cell contains 20 transistors of varying channel height and width, allowing great flexibility. Transistors can be connected in parallel to approximate other channel widths. Fast configuration, good I/O.
Evolvable Motherboard [43] - Array of analog switches, connected to six interchangeable evolvable units. Evolution of gates, amplifiers and oscillators demonstrated using bipolar transistors as evolvable unit. Good I/O. Board-based architecture is not suitable for real world problems due to size, cost and number of evolvable units.
FIPSOC [65] - Complete evolutionary system aimed at mixed signal environments. Analog and digital units that can undergo evolution. CPU and memory to encode evolutionary algorithm. Analog units based around amplifiers. Digital units based on LUTs and flipflops. Context-based dynamic reconfiguration suitable for real-time adaptive systems.
Complete Hardware Evolution [91] - FPGA-independent evolutionary system on a chip consisting of a genetic algorithm pipeline including evaluation, functional design and storage for population. Limited to small populations of small chromosomes. Large FPGAs required.

4 Case Studies

The final section of this paper presents two case studies – one evolved at a logic netlist level and evaluated extrinsically, and one evolved at a device specific netlist level, evaluated intrinsically. The problem selected for both was the evolution of a two bit adder. This problem has been well studied in the past, initially by Louis and Rawlins [51] and more recently by Miller, Thomson and Fogarty [62], Coello and Aguirre [7] and Hollingworth and Tyrell [29].

4.1 Case Study 1 – Logic Level Evolution

4.1.1 Phenotype Abstraction

In this experiment we evolve the adder at a reasonably high level of abstraction. The representation is restricted to combinational digital circuits only. The minimum reconfigurable structural units are at the level of gates - we allowed AND, OR , NOT, XOR, and multiplexer gates. The selection of these resources was based loosely on the resources available in the Xilinx 6200 series FPGA architecture. This means we have abstracted away any of the physics in the device and consider only logical solutions, which limits the opportunity for innovation. However we have made the search space a lot smaller than if such features were included. This abstraction allows fast simulation of the evaluation function. If mapping rules are followed properly and the circuit is operated within the constraints set by the manufacturer of the technology it is mapped onto, it will function as it did in the abstracted simulation.

The circuit abstraction used in the first case study is based on that used by Miller et al. [62] to investigate the evolution of combinational circuits. The circuit is represented by a numbered rectangle of cells. They are indexed from the top left cell, row-wise then column-wise. Each cell has two inputs and one logic function. The function may either be a two input logic gate or a multiplexer. Inputs to a cell can be either from other cells, or the circuit inputs. The circuit inputs are the test input vector, the inverted test input vector, logic 0 or logic 1. To avoid feedback, each input must be from a cell with a lower number than the cell itself. The circuit outputs required are restricted to the top and right hand side of the cell array.

4.1.2 Genetic Algorithm and Genotype

The circuit is presented to the algorithm as an integer string. There is a triplet of integers for each cell, representing the sources of the two cell inputs and the cell function, with the triplet locus mapping to the cell index. If the cell function is a logic gate, the function allele represents a specific logic gate. If the cell function is a multiplexer, then the allele represents the multiplexer control signal source,

which can be either the output of another cell or a circuit input. The list of functions used is shown below. Cell outputs are represented by an integer each, the allele referring to the output cell index.

The genetic algorithm used was a standard population-based linear genetic algorithm with selection, crossover and mutation operators, after Goldberg [15]. However the allele range of each gene varied. Therefore it was required that the cardinality of each gene can be set independently of the others. On application to a gene the mutation operator selected a random integer between zero and the cardinality. No bias was placed on selecting integers similar to the current value, as in most cases the relationship between the alleles in the context of this problem was not clear, and so the ordering of the alleles was arbitrary.

Fitness was measured by subjecting each candidate circuit to a test vector containing the complete two bit adder truth table. One fitness point was awarded for each correct bit in the output sequence, giving a total of 96 for maximum fitness.

The circuit inputs were A0, A1, B0, B1, Carry In, !A0, !A1, !B0, !B1, !Carry In, 0, and 1. The outputs were S0, S1, and Carry Out.

Allele	Function	Allele	Function
0	A . B	7	!A
1	A . !B	8	!A +B
2	!A . B	9	!B
3	A ⊕ B	10	!A + B
4	A + B	11	!A + !B
5	!A . !B	12....	!C . A + C . B, C = circuit input 0
6	!A ⊕ B	...(n)	!C . A + C . B, C = input (n-12)

Table 2: List of function to allele mappings used.

Following informal experiments the genetic parameters were mostly chosen by selecting those that Miller had found useful in [62]. Uniform crossover was used with breeding rate at 100%. The mutation rate was set to 5% of all genes in the population. The population size was set to 30. Qualitative examination of early runs led us to believe that the suggested 40,000 - 80,000 generations were not necessary to achieve good results, with 20,000 generations sufficing. Two-member tournament selection was used. A tournament selection pressure as described by Miller was also introduced, and set to 0.7, meaning that the winner of each tournament was selected only with 70% probability. The first generation of each run was randomly generated.

4.1.3 Results

10 runs were each made for cell array sizes of 3 x 3, 3 wide by 4 high, and 4 x 4. Overall results are shown below, and relate well to the results presented by Miller. Fitnesses and deviations have been scaled to 100. Each evolutionary run took around four minutes to complete on a 433MHz Celeron CPU.

Array Size	Mean fitness of Best Solutions	Std. Dev. Of Best Solutions	% of Runs with Perfect Solutions
3x3	96.25	4.99	50%
4x3	96.88	3.71	50%
4x4	97.50	4.03	60%

Table 3: Results from 10 runs of 2 bit adder evolution across a range of array sizes

A series of 10 random searches were also carried out on the 3x3 array search space, using the same number of evaluations as the evolutionary runs. These are shown in Fig. 5, along with the best fitnesses of each generation of the 3x3 array evolutionary runs.

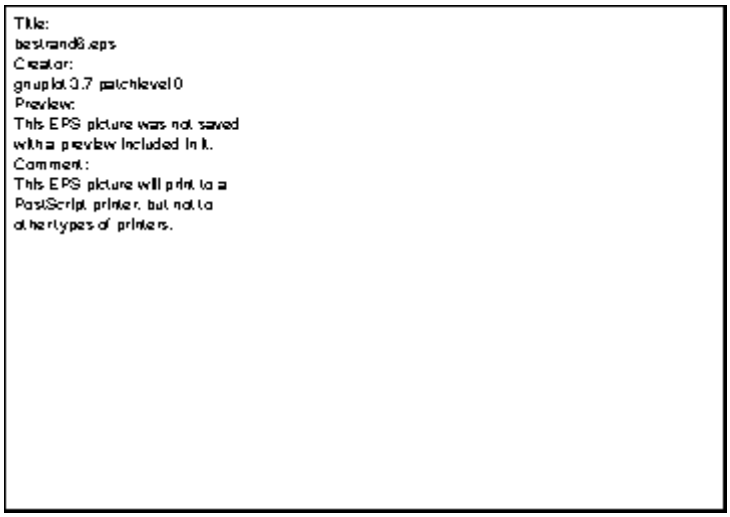

Fig. 5: Best fitness against generations of an evolved 2 bit adder on a 3x3 array along with 10 random searches of the same length.

4.1.4 Discussion

The results achieved from these experiments broadly agree with Miller's results. For example, for a 3x3 array, Miller reported a mean fitness of 96.14 with a standard deviation of 4.52, and 50% of cases perfect.

An example of a two bit adder evolved on the 3x3 array is show n in Fig. 6. It can be seen that this is a minor variation on a traditional two bit ripple-carry adder [26].

Note that this design uses XOR gates to generate the sum signals, and XOR and multiplexer gates to generate both carry signals. We have already noted that it is not trivial for human designers to develop efficient circuits that use only XOR and multiplexer gates. Thus evolution is demonstrated as a means for successfully searching a poorly understood design space for an efficient (in terms of gate count) circuit. Miller et al. have presented similar circuit designs.

The results display a trend towards higher mean fitnesses for larger arrays. This suggests that it is easier to find perfect solutions in larger arrays. However the solutions found within these arrays tended to involve more gates and connections than those discovered on smaller arrays, and so in this respect are less efficient. This is not surprising as no bias was used to search for parsimonious solutions. Miller et al also noted such trends.

There were some minor differences between the algorithm and that reported by Miller. Miller used a "levels back" parameter to limit the length of the routing connections to a certain number of columns to the left of the current cell. When using such small arrays, we felt this was not necessary. Miller also ran his algorithm for more generations. Informal examination of our early results showed no or little improvement from running more than 20,000 generations. Miller also used a slightly different scheme to map multiplexer behavior to the chromosome.

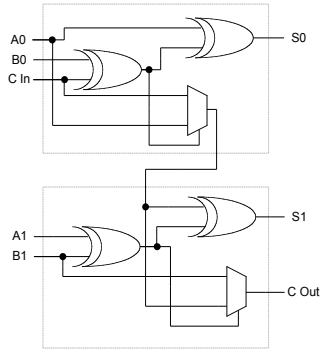

Fig. 6: Example of an adder evolved on a 3x3 grid.

4.2 Case study 2: Device Specific Netlist Level Evolvable hardware

For our intrinsic case study we selected the Xilinx Virtex FPGA. Of current commercial FPGA architectures, it fitted the criteria for a general evolvable hardware platform specified above better than any others. It is also a relatively new architecture with a long life ahead of it, so architecture-specific findings will be of interest for some time to come.

4.2.1 The Virtex Architecture

The SRAM-based Virtex can be reconfigured infinitely. With up to 60 MBs^{-1} of configuration bandwidth, it can also be configured quickly. Although some of the bitstream is proprietary and undisclosed, most hardware resources can be configured in Java with the Xilinx JBits software. It allows partial reconfiguration to reduce configuration time, and is widely available at reasonable cost.

A simplified diagram of the Virtex architecture is shown in Fig. 7. It is arranged as an array of configurable logic blocks (CLBs). There are three categories of routing between CLBs - single lines that connect CLBs to their neighbors, hex lines that connect a CLB to CLBs six blocks away, and long lines. I/O blocks providing logic and drivers for several I/O standards, surround the CLB array.

Each CLB consists of two slices, which each contain two SRAM-based four input function lookup tables (LUTs), some fast carry logic, and two flip-flops. The inputs to each CLB slice consist of thirteen inputs. These are the inputs to both LUTs, and flip-flop clock, data, enable and set/reset lines. Outputs from each CLB can be drawn from a range of internal signals, including the lookup tables, the carry logic and the flip-flops. Configurable multiplexers select the connection between the routing and the CLB inputs and outputs.

The main problem with Virtex is that it can be damaged by invalid configurations. Each CLB is driven independently, so it is possible for contention to arise between two drivers if two output multiplexers from two different CLBs drive the same line.

Experiments showing the feasibility of Virtex as a reconfigurable platform were first presented by Hollingworth et al. [27]. JBits was used to map circuit designs to the configuration bitstream. Contention has been avoided by using fixed, handcrafted routing between the CLBs, initially only in a feed forward structure, but more recently by imposing a 6200-like sea of gates structure where one routing signal for each neighbor is allowed in and out of each CLB [28]. Levi has also used Virtex for evolutionary experiments [47], but the method of contention avoidance has not been reported.

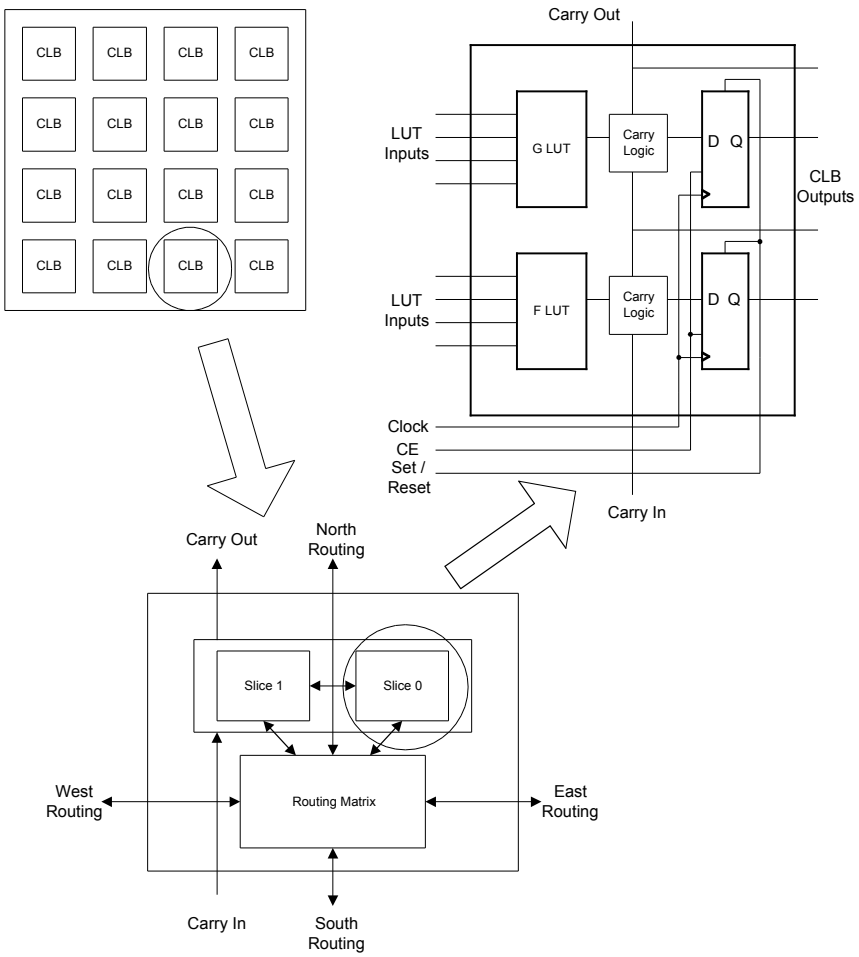

Fig. 7: A simplified diagram of the Virtex Architecture

4.2.2 Genotype Abstraction

As discussed earlier, the most innovative design space of circuits that can be represented using an FPGA is that provided by the configuration bitstream itself – the finest level of reconfigurability of the FPGA. Although the Virtex architecture is course grained it can be configured at a fine level of detail. Unfortunately a good deal of the bitstream-to-architecture mapping is proprietary knowledge. However Xilinx JBits software allows us to reconfigure at a fairly low level of detail, that of a netlist specific to the Virtex device with timing, routing and

geometry included in the abstraction. With this we were able to encode almost all the resources available on the hardware, thereby allowing evolution to search a large space for innovative solutions.

To capture the netlist level abstraction correctly in the genotype, we opted to avoid using a binary representation. This was mainly due to our desire to include as many of the resources available to us as possible. First, if all possible configurations were encoded as a series of unique binary strings, arbitrary bias would be introduced against resources that spanned several bits. Alternatively, if each fine-grain resource were separately encoded as a number of bits, arbitrary redundancy would be introduced when the number of states that resource could assume (i.e. number of alleles) is not a power of two. This would also introduce unknown biases.

For these reasons it was decided that each resource that could be modified by the JBits API would be encoded as a separate integer gene. (The exception to this was the LUT configurations, which were encoded as sixteen bits.)

Routing representation was also an issue. It was so important to avoid damage of the device due to contention that we restricted the representation to avoid these areas of space. In more detail, only the single (nearest neighbor) wires were evolved. It was noted that although CLB input multiplexers can connect to a wide range of single lines, the connections between the output multiplexers and singles are sparse, and few can connect to any that their neighbors can. In fact, only eight of the possible forty-eight connections had to be prohibited to prevent any possible contention arising. The connection points between routing wires were not evolved. An overview of the chromosome structure for one CLB and its associated routing is shown in Table 4.

Number of Genes	Type of Gene	Alleles / gene (Cardinality)
16	LUT Input MUXes	27
2	Clock Input MUXes	11
2	SR Input MUXes	10
2	Clock Enable Input MUXes	11
4	Other Input MUXes	4
64	LUT Configuration	2
48	Other CLB Logic	1-3
8	Output signal MUX	13
40	Output MUX to single switches	2

Table 4: Overview of the genotype for one CLB.

A small rectangle of the chip was selected for evolution. The cells on the edges of this area were further restricted. They were only allowed to use the routing

connections to the rest of the evolved area or circuit input wires, and not outside. Inputs were fed directly into LUTs on the west edge. For instance all four LUTs on the southwest corner were restricted to using the A0 and B0 inputs, the carry input and the remaining input for each LUT was restricted to west and north connections. All other inputs to the CLB were also restricted to the west and north. The output of the evolved area was taken from hex lines on the west edge, which were not evolved. This led to a chromosome of 604 genes for a 2x2 evolved area. A diagram of the inputs to and outputs from the evolved area is shown in Fig. 8.

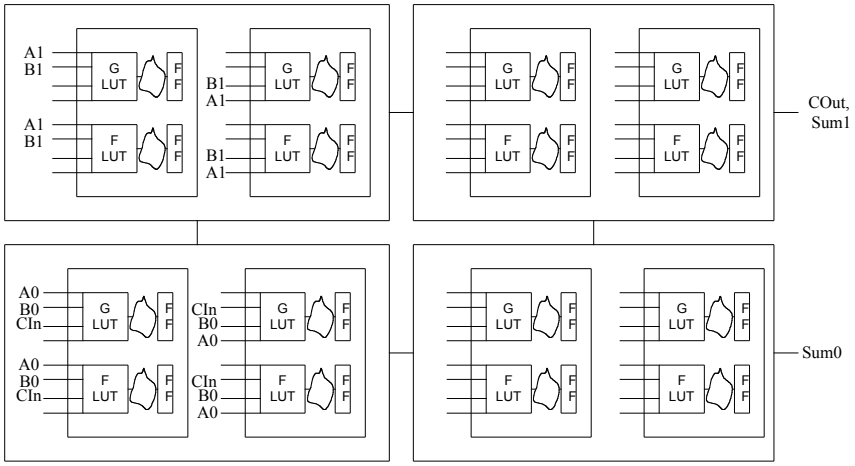

Fig. 8: Inputs and outputs to the evolved area. Unlabelled LUT inputs were completely under evolutionary control.

One-point crossover was used as opposed to uniform, following recent work by Vassilev and Miller [94] on the suitability of uniform crossover to three bit multiplier landscapes. The population size was also increased to 50. All other parameters were set as in the first experiment.

Unlike the experiment in simulation described in the first case study, the solutions were not constrained to combinational circuits, as the representation allowed a feedback. Working at low abstractions it may be necessary to specify generalization that is normally hidden by higher abstractions, as discussed in section 3.3. In order to ensure generalization across any order of input sequences the order of presentation was randomized for each evaluation. The genetic representation allows for circuits that may not generate the same fitness when evaluated twice - the outputs may exhibit dynamical variations unrelated to the inputs. Although such circuits are not useful final solutions they may contain valuable information about how to solve part of the problem, or how to traverse the fitness landscape. Because of this each individual was evaluated five times, and its worst fitness selected rather than discarding these solutions.

Initial experiments suffered from the speed it takes to reconfigure the FPGA. With a bitstream of over 300k configuration of each individual took around 1 second. Consequently the number of evaluations that could be carried out in a reasonable timeframe were low.[2] The work in simulation by both Miller and us suggested that large numbers of evaluations are needed, and this was confirmed by failure to evolve fully functional two bit adders within 1000 generations of two test runs. As we were evolving only a small area of the FPGA we had two options open to us – either batch multiple evaluations together in one configuration or use partial reconfiguration to reduce the bus traffic. As discussed earlier the most elegant solution to this problem is the use of partial reconfiguration. This is the approach we took, giving us an evaluation speedup of an order of magnitude. Simulation in the much more abstract logical space used in the first case study was still faster to complete the genotype-phenotype map and evaluation of the intrinsic evolution.

4.2.3 Results

10 runs were carried out for a 2x2 cell. Overall results are shown below. Fitnesses and deviations have again been scaled to 100. The mean number of generations to find a perfect solution and the corresponding standard deviation ignore the run that did not find a perfect solution. The average time to find a perfect solution was around 3 hours and 45 minutes using a 433MHz Celeron PC.

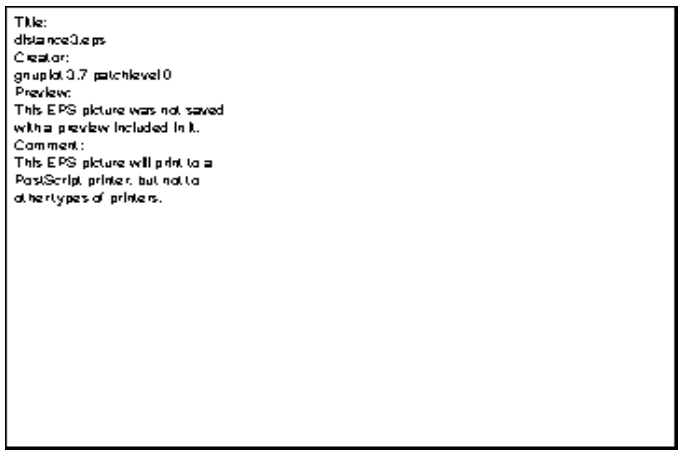

Fig. 9: Best fitness and average Euclidean distance between members of the population against generations for the first 1000 generations of the initial run.

[2] Note that this highlights the misconception that extrinsic evaluation is slower than intrinsic. If a logical abstraction is used, evaluation in simulation on a PC can be very fast as the microprocessor is designed to carry out fast logical calculations.

Mean Fitness of Best Solutions	Std. Dev. of Best Solutions	% of Runs With Perfect Solutions	Mean Gens to Find Perfect	St. Dev of Gens to Find Perfect
99.58	1.26	90	2661	2169

Table 5: Results from 10 runs of intrinsic 2 bit adder evolution

4.2.4 Discussion

The results achieved from these experiments show an improvement in all statistics over the results carried out in simulation, even though the size of the space that the search was conducted across is much greater. Hence the combination of biases mean this space is more tractable to search. However it is difficult to say much more than this. The biases of the two experiments are very different - there were variations in representation, mapping and ordering biases. One point that should be made is that the Virtex architecture is rich in resources suitable for creating adders, which is likely to help evolvability. The LUT is a very useful primitive for generating logical behaviors, especially when the behavior is specified in truth tables. Additionally, logic is present within each Virtex CLB intended for performing logical carrys. In the spirit of working at a low level of abstraction that we have advocated during this discussion, our representation allows much of this logic to be manipulated by the algorithm.

Hollingworth that Tyrell have also evolved two bit adders (but without carry) intrinsically using Virtex [28]. In this case they used a fixed feed-forward routing structure and evolved only the bits of six LUTs. Such a representation cannot make use of the additional carry logic, which many of the two bit adders evolved here do. Hence we see that relaxing abstraction constraints may facilitate design innovation as discussed in section 3.2.

We should expect that such relaxations mean more work for the algorithm. Comparison with Hollingworth that Tyrell's results support this. They evolved a population of 100 chromosomes using a genetic algorithm with one point crossover and mutation. Their results showed a 100% success rate, and a mean and standard deviation of 808 and 259 generations respectively to find a perfect solution. What is most revealing is that such a huge increase in space does not appear to have been accompanied by a huge decrease in algorithm performance, rather only a moderate decrease. It was noted in section 3.4 that evolvability does not necessarily decrease with relaxation of constraints, and in some cases increases. So again this can be taken as a sign that working in low abstraction spaces should be advocated.

However it should be noted that adder circuits are not the best problems to demonstrate innovation over a sequential search space. Such a space is more suited to problems that make use of the range of behaviors over the time

dimension opened by the relaxation in abstraction, such as Thompson's tone discriminator [80]. More complex problems such as this are also likely to benefit from the use of dynamic representational biases such as developmental processes.

This experiment was carried out using parameters in line with neutral network theory, which was discussed in section 3.4. It can be seen from Fig. 9 that evolution continues long after the initial convergence of the population, which in this example occurs within the first few generations. In addition, improvements take place after long periods of no improvement, which may correspond to periods when the population is moving across neutral planes or ridges. These improvements are also associated with a sudden convergence of the population, which then gradually diverges again. Our observations are in line with the hypothesis that neutral networks permeate this kind of search space.

5 Summary

The problems of electronic circuit design are increasing as demand for improvements increases. In this review we have introduced a promising new type of solution to these difficulties - evolvable hardware. This emerging field exists at the intersection of electronic engineering, computer science and biology. The benefits brought about by evolvable hardware are particularly suited to a number of applications, including the design of low cost hardware, poorly specified problems, creation of adaptive systems, fault tolerant systems and innovation.

As research in this field accelerates, new methods of classifying the many strands of research must be found. Here we identified three viewpoints: the level of abstraction, the bias implementation and the hardware evaluation process. With these in mind, current research trends in evolvable hardware were reviewed and analyzed. In particular, the research focusing on innovation, evolvability and platforms were described.

Finally, this work presented two case studies - one evolved at a logic netlist level and evaluated extrinsically, and one evolved at a device specific netlist level, evaluated intrinsically. The problem selected for both was the evolution of a two bit adder. These demonstrated the ability of evolution to successfully find solutions using a low abstraction and illustrated how such an approach can allow innovation in addition to automation.

Evolvable hardware is still a young field. It does not have all the answers to the problems of circuit design and there are still many difficulties to overcome. Nevertheless, these new ideas may be one of the brightest and best hopes for the future of electronics.

Acknowledgements

The authors would like to thank Dr. Peter Rounce for his insights and advice. Timothy Gordon is supported by an EPSRC studentship.

References

[1] Andersen P. (1998) Evolvable Hardware*: Artificial Evolution of Hardware Circuits in Simulation and Reality*, M.Sc. Thesis, University of Aarhus, Denmark.

[2] Arslan T. and Horrocks D.H. (1995), The Design of Analogue and Digital Filters Using Genetic Algorithms, *Proc. of the 15th SARAGA Colloquium on Digital and Analogue Filters and Filtering Systems*, pp. 2/1 – 2/5, London, U.K.

[3] Banzhaff W., Nordin P., Keller E. and Francone F.D. (1998*), Genetic Programming*, Morgan-Kaufmann, San Francisco, CA, U.S.A.

[4] Bentley P.J. and Kumar S. (1999), Three Ways to Grow Designs: A Comparison of Evolved Embryogenies for a Design Problem, *Proc. of the Genetic and Evolutionary Computation Conf.*, Orlando, FL, U.S.A., pp.35-43.

[5] Brooks R. A. (1991), Intelligence without representation*, Artificial Intelligence J.*, **47**, pp. 139-159.

[6] Cliff D., Harvey I., and Husbands P. (1993), Explorations in Evolutionary Robotics, *Adaptive Behaviour*, **2**, 1, pp.73-110.

[7] Coello Coello C.A., Christiansen A.D and Hernández Aguirre A. (2000), Using Evolutionary Techniques to Automate the Design of Combinational Circuits, *International Journal of Smart Engineering System Design*, **2**, no. 4, pp. 229-314.

[8] Damiani E., Liberali V. and Tettamanzi A.G.B. (2000), Dynamic Optimisation of Non-linear Feed-Forward Circuits*, Proc. of the 3rd Int. Conf. on Evolvable Systems*, Edinburgh, U.K., pp. 41-50.

[9] de Garis H. (1994), An Artificial Brain: ATR's CAM-Brain Project Aims to Build/Evolve an Artificial Brain with a Million Neural Net Modules Inside a Trillion Cell Cellular Automata Machine, *New Generation Computing J.*, **12**, no. 2, pp. 215-221.

[10] de Jong, K.A. (2001), *Evolutionary Computation*, MIT Press, Cambridge MA, U.S.A.

[11] Flockton S.J. and Sheehan K. (1999), A system for Intrinsic Evolution of Linear and Non-linear Filters, *Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, Pasadena, CA, U.S.A, pp.93-100.

[12] Fukunaga A. and Stechert A. (1998), Evolving Nonlinear Predictive Models for Lossless Image Compression with Genetic Programming, *Proc. of the 3rd Annual Genetic Programming Conf.*, Madison, WI, U.S.A, pp. 95-102.

[13] Girau B., Marchal P., Nussbaum P. and Tisserand A. (1999), Evolvable Platform for Array Processing: A One-Chip Approach, *Proc. of the 7th Int. Conf. on Microelectronics for Neural, Fuzzy and Bio-inspired Systems*, Granada, Spain, pp. 187-193.

[14] Göckel N., Drechsler R. and Becker B. (1997), A Multi-Layer Detailed Routing Approach based on Evolutionary Algorithms, *Proc. of the IEEE Int. Conf. on Evolutionary Computation*, Indianapolis, IN, U.S.A., pp. 557-562.

[15] Goldberg D.E. (1989*), Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, MA, U.S.A.

[16] Goldberg D.E., Deb K. and Korb B. (1991), Don't Worry, Be Messy, *Proc. of the 4th Int. Conf. on Genetic Algorithms and their Applications*, San Diego, CA, U.S.A., pp. 24-30.
[17] Gordon D.F. and des Jardins M. (1995), Machine Learning *J.*, **20**, pp. 1-17.
[18] Grimbleby J.B. (2000), Automatic Analogue Circuit Synthesis Using Genetic Algorithms, *IEE Proc. - Circuits Devices, Systems*, **147**, no. 6, pp. 319-323.
[19] Hamilton A., Papathanasiou K., Tamplin M. and Brandtner T. (1998), Palmo: Field Programmable Analogue and Mixed-Signal VLSI for Evolvable Hardware, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp. 335-344.
[20] Harvey I. (1991), Species Adaptation Genetic Algorithms: The basis for a continuing SAGA*, Proc. of the 1st European Conf. on Artificial Life*, Paris, Franc, pp. 346-354.
[21] Harvey I and Thompson A. (1996), Through the Labyrinth Evolution Finds a Way: A Silicon Ridge, *Proc. of the 1st Int. Conf. on Evolvable Systems*, Tsukuba, Japan, pp. 406-422.
[22] Hemmi H., Mizoguchi J and Shimohara K. (1996), Evolving Large Scale Digital Circuits, *Proc. of the 5th Int. Workshop on the Synthesis and Simulation of Living Systems*, Nara, Japan, pp. 168-173.
[23] Higuchi T., Iba H., and Manderick B. (1994), Evolvable Hardware, in *Massively Parallel Artifical Intelligence*, MIT Press, Cambridge, MA, U.S.A., pp. 398-421.
[24] Higuchi T., Iwata M., Kajitani I., Iba H., Hirao Y., Manderick B., and Furuya T. (1996), Evolvable Hardware and its Applications to Pattern Recognition and Fault-tolerant Systems, in *Towards Evolvable Hardware: The Evolutionary Engineering Approach*, Sanchez E. and Tomassini M. (Eds.), Springer-Verlag, Berlin, Germany, pp. 118-135.
[25] Hirst A.J. (1996), Notes on the Evolution of Adaptive Hardware, *Proc. of Adaptive Computing in Engineering Design and Control*, Plymouth, U.K., pp. 212-219.
[26] Holdsworth B. (1993) *Digital Logic Design*, Butterworth-Heinemann, Oxford, UK.
[27] Hollingworth G., Smith S. and Tyrrell A. (2000), The Safe Intrinsic Evolution of Virtex Devices, *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, Palo Alto, CA, U.S.A.
[28] Hollingworth G., Smith S. and Tyrell A. (2000), The Intrinsic Evolution of Virtex Devices Through Internet Reconfigurable Logic, *Proc. of the 3rd Int. Conf. on Evolvable Systems*, Edinburgh, U.K., pp. 72-79,
[29] Huynen M.A., Stadler P.F. and Fontana W. (1996), Smoothness within ruggedness: The role of neutrality in adaptation, *Proc. of the National Academy of Science*, **93**, 397-401.
[30] Heyworth K. (1998), The "Modeling Clay" Approach to Bio-inspired Electronic Hardware, *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware*, Lausanne, Switzerland, pp. 248-255.
[31] Imamura K., Foster J. A. and Krings A.W. (2000), The Test Vector Problem and Limitations to Evolving Digital Circuits, *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, Palo Alto, CA, U.S.A, pp.75-79.
[32] Iwata M., Kajitani I., Yamada H., Iba H. and Higuchi T. (1996), A Pattern Recognition System Using Evolvable Hardware, *Proc. of the 4th Int. Conf. on Parallel Problem Solving from Nature*, Berlin, Germany, pp. 761-770.
[33] Job D., Shankararaman V. and Miller J.F. (1999), Hybrid AI Techniques for Software Design, *Proc. of the 11th Int. Conf. on Software Engineering and Knowledge Engineering*, Kaiserslautern, Germany, pp. 315-319.
[34] Kajitani I., Hoshino T., Iwata M. and Higuchi T. (1996), Variable length chromosome GA for Evolvable Hardware, *Proc. of the 3rd Int. Conf. on Evolutionary Computation*, Nagoya, Japan, pp. 443-447.

[35] Kajitani I., Hoshino T., Nishikawa D., Yokoi H., Nakaya S., Yamauchi T., Inuo T., Kajihara N., Iwata M., Keymeulen D. and Higuchi T. (1998), A Gate-Level EHW Chip: Implementing GA Operations and Reconfigurable Hardware on a Single LSI, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp. 1-12.

[36] Kalganova T., Miller J.F. and Lipnitskaya N., (1998), Multiple Valued Combinational Circuits Synthesised using Evolvable Hardware Approach, *Proc. of the 7th Workshop on Post-Binary Ultra Large Scale Integration Systems*, Fukuoka, Japan.

[37] Kalganova T. and Miller J.F. (1999), Evolving More Efficient Digital Circuits by Allowing Circuit Layout Evolution and Multi-Objective Fitness*, Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, Pasadena, CA, U.S.A, pp. 54-63.

[38] Kauffman S. and Levin S. (1987), Towards a General Theory of Adaptive Walks on Rugged Landscapes, *J. of Theoretical Biology*, **128**, pp.11-45.

[39] Keymeulen D., Iwata M., Kuniyoshi Y. and Higuchi T. (1998), Comparison between an Off-line Model-free and an On-line Model-based Evolution applied to a Robotics Navigation System using Evolvable Hardware, *Proc. of the 6th Int. Conf. on Artificial Life*, Los Angeles, CA, U.S.A. pp.109-209.

[40] Kitano H. (1998), Building Complex Systems Using Developmental Process: An Engineering Approach, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp. 218-229.

[41] Koza J., Bennett F. H, III, Andre D., Keane M.A. (1999), *Genetic Programming III*, Morgan-Kauffmann, San Francisco, CA, U.S.A.

[42] Langeheine J., Folling S., Keir K., Schemmel J. (2000), Towards a Silicon Primordial Sourp: A Fast Approach to Hardware Evolution with a VLSI Transistor Array, *Proc. of the 3rd Int. Conf. on Evolvable Systems*, Edinburgh, U.K., pp. 123-132.

[43] Layzell P. (1999), Reducing Hardware Evolution's Dependency on FPGAs**,** *Proc. of the 7th Int. Conf. on Microelectronics for Neural, Fuzzy and Bio-inspired Systems*, Granada, Spain, pp. 171-178

[44] Layzell P. (1999), Inherent Qualities of Circuits Designed by Artificial Evolution: A Preliminary Study of Populational Fault Tolerance, *Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, Pasadena, CA, U.S.A, pp. 85-86.

[45] Layzell P. and Thompson A. (2000), Understanding Inherent Qualities of Evolved Circuits: Evolutionary History as a Predictor of Fault Tolerance, *Proc. of the 3rd Int. Conf. on Evolvable Systems*, Edinburgh, U.K., pp. 133-144.

[46] Levi D. and Guccione S.A. (1999), GeneticFPGA: Evolving Stable Circuits on Mainstream FPGA Devices, *Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, Pasadena, CA, U.S.A, pp. 12-17.

[47] Levi D. (2000), HereBoy: a fast evolutionary algorithm, *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, Palo Alto, CA, U.S.A, pp. 17-24.

[48] Liu W., Murakawa M., and Higuchi T. (1996), ATM Cell Scheduling by Function Level Evolvable Hardware, *Proc. of the 1st Int. Conf. on Evolvable Systems*, Tsukuba, Japan, pp. 180-192.

[49] Lohn J.D. and Columbano S.P. (1998), Automated Analog Circuit Synthesis Using a Linear Representation, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp. 125-133.

[50] Lohn J.D., Haith G.L., Colombano S.P. and Stassinopoulos D. (1999), A Comparison of Dynamic Fitness Schedules for Evolutionary Design of Amplifiers, *Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, Pasadena, CA, U.S.A, pp.87-92.

[51] Louis S.J. and Rawlins G.J.E. (1991), Designer Genetic Algorithms: Genetic Algorithms in Structure Design, *Proc. of the 4th Int. Conf. on Genetic Algorithms*, San Diego, CA, U.S.A., pp. 53-60.

[52] Manovit C, Aporntewan C. and Chongstitvatana P. (1998), Synthesis of Synchronous Sequential Logic Circuits from Partial Input/Output Sequences, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp.98-105..

[53] Masner J., Cavalieri J., Frenzel J. and Foster J., Representation and Robustness for Evolved Sorting Networks*, Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, Pasadena, CA, U.S.A, pp.255-261.

[54] Mazumder P. and Rudnick E. M. (1999), *Genetic Algorithms for VLSI Design, Layout and Test Automation*, Prentice-Hall, Upper Saddle River, NJ, U.S.A.

[55] Miller J.F., Kalganova T., Lipnitskaya N. and Job D. (1999), The Genetic Algorithm as a Discovery Engine: Strange Circuits and New Principles, *Proc. of the AISB Symposium on Creative Evolutionary Systems*, Edinburgh, U.K, pp. 65-74.

[56] Miller J.F., Job D. and Vassilev V. K. (2000) Principles in the Evolutionary Design of Digital Circuits – Part I*, Genetic Programming and Evolvable Machines*, **1**, no. 1/2, pp. 7-35.

[57] Miller J.F., Job D. and Vassilev V. K. (2000) Principles in the Evolutionary Design of Digital Circuits – Part II*, Genetic Programming and Evolvable Machines*, **1**, no. 3, pp. 259-288.

[58] Miller J.F. and Thomson P. (1995), Combinational and Sequential Logic Optimisation using Genetic Algorithms, *Proc. of the 1st Int. Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications*, Sheffield, U.K., pp. 34-38.

[59] Miller J.F. and Thomson P. (1998), Aspects of Digital Evolution: Evolvability and Architecture, *Proc. of the 5th Int. Conf. on Parallel Problem Solving in Nature*, Amsterdam, The Netherlands, pp. 927-936.

[60] Miller J.F. and Thomson P. (1998), Aspects of Digital Evolution: Geometry and Learning, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp.25-35.

[61] Miller J.F. and Thomson P. (1998), Evolving Digital Electronic Circuits for Real-Valued Function Generation using a Genetic Algorithm*, Proc. of the 3rd Annual Conf. on Genetic Programming*, San Francisco, CA, U.S.A, pp. 863-868.

[62] Miller J.F., Thomson P. and Fogarty T.C. (1997), Designing Electronic Circuits using Evolutionary Algorithms. Arithmetic Circuits: A Case Study, in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications*, Quagliarella D., Periaux J., Poloni C. and Winter G. (Eds.), John Wiley & Sons, London, UK.

[63] Mitchell M. (1998), *An Introduction to Genetic Algorithms*, MIT Press, Cambridge MA, U.S.A.

[64] Mitchell T.M. (1997), *Machine Learning*, McGraw-Hill, London, UK

[65] Moreno J.M., Madrenas J., Faura J., Canto E., Cabestany J. and Insenser J.M. (1998), Feasible Evolutionary and Self-repairing Hardware by Means of the Dynamic Reconfiguration Capabilities of the FIPSOC Devices, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp.345-355.

[66] Murakawa M., Yoshizawa S., Adachi T., Suzuki S., Takasuka K., Iwata M. and Higuchi T. (1998), Analog EHW Chip for Intermediate Frequency Filters, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp. 134-143.

[67] Murakawa M., Yoshizawa S., Kajitani I., Furuya T., Iwata M., and Higuchi T. (1996), Hardware Evolution at Function Level, *Proc. of the 4th Conf. on Parallel Problem Solving from Nature*, Berlin, Germany, pp. 62-71.

[68] Murakawa M., Yoshizawa S., Kajitani I., Yao X., Kajihara N., Iwata M. and Higuchi T. (1999), The GRD Chip: Genetic reconfiguration of DSPs for Neural Network Processing, *IEEE Trans. on Computers*, **48**, no. 6, pp. 628-639.

[69] Ortega C. and Tyrrell A. (1999), Biologically Inspired Fault-tolerant Architectures for Real-time Control Applications, *Control Engineering Practice*, **7**, no. 5, pp. 673-678.
[70] Pollack J.B., Lipson H., Ficici S., Funes P., Hornby G. and Watson R. (2000), Evolutionary Techniques in Physical Robotics, *Proc. of the 3rd Int. Conf. on Evolvable Systems*, Edinburgh, U.K., pp. 175-186.
[71] Rendell, L. (1987), Similarity-based Learning and its Extensions, *Computational Intelligence*, **3**, pp.241-266.
[72] Rosenman, M. (1997), The Generation of form Using an Evolutionary Approach, in *Evolutionary Algorithms in Engineering Applications,* Dasgupta D. and Michalewicz (Eds) Springer-Verlag, pp. 69-86.
[73] Rumelhart, D.E., Widrow B and Lehr M. (1994). The Basic Ideas in Neural Networks*, Communications of the ACM*, **37**, no. 3, pp. 87-92.
[74] Salami M., Murakawa M. and Higuchi T. (1996), Data Compression based on Evolvable Hardware, *Proc. of the 1st Int. Conf. on Evolvable Systems*, Tsukuba, Japan, pp. 169-179.
[75] Salami M., Sakanashi H., Tanaka M., Iwata M., Kurita T. and Higuchi T. (1998), On-Line Compression of High Precision Printer Images by Evolvable Hardware, *Proc. of the Data Compression Conf.*, Los Alamitos, CA, U.S.A. pp. 219-228.
[76] Sechen C. (1988*), VLSI Placement and Global Routing Using Simulated Annealing*, Kluwer Academic Publishers, Boston MA, U.S.A.
[77] Stoica A., Fukunaga A., Hayworth K. and Salazar-Lazaro C. (1998), Evolvable Hardware for Space Applications, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp. 166-173.
[78] Stoica A., Keymeulen D., Tawel R., Salazar-Lazaro C., Li W. (1999), Evolutionary Experiments with a Fine-Grained Reconfigurable Architecture for Analog and Digital CMOS Circuits, *Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, Pasadena, CA, U.S.A, pp.76-85.
[79] Stoica A., Zebulum R. and Keymeulen D., (2000), Mixtrinsic Evolution, *Proc. of the 3rd Int. Conf. on Evolvable Systems*, Edinburgh, U.K., pp. 208-217.
[80] Thompson A. (1996), Silicon Evolution, *Proc. of the 1st Annual Conf. on Genetic Programming*, Stanford, CA, U.S.A., pp. 444-452.
[81] Thompson A. (1997), Evolving Inherently Fault-Tolerant Systems, *Proc. of Institution of Mechanical Engineers*, **211**, part I, pp.365-371
[82] Thompson A. (1998), *Hardware Evolution*, Springer-Verlag, London, U.K.
[83] Thompson A. (1998), On the Automatic Design of Robust Electronics Through Artificial Evolution, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp.13-25.
[84] Thompson A., Harvey I. and Husbands P. (1996), Unconstrained Evolution and Hard Consequences, *in Towards Evolvable Hardware: the evolutionary engineering approach*, Sanchez E. and Tomassini M. (Eds.), Springer-Verlag, Berlin, Germany, pp. 136-165.
[85] Thompson A. and Layzell P. (1999), Analysis of Unconventional Evolved Electronics, *Communications of the ACM*, **42**, 4, pp. 71-79.
[86] Thompson A. and Layzell P. (2000), Evolution of Robustness in an Electronics Design, *Proc. of the 3rd Int. Conf. on Evolvable Systems*, Edinburgh, U.K., pp. 218-228.
[87] Thompson A. and Wasshuber C. (2000), Design of Single Electron Systems through Artificial Evolution, *Int. J. of Circuit Theory and Applications*, **28**, no. 6, pp. 585-599.
[88] Torresen J. (1998), A Divide and Conquer Approach to Evolvable Hardware, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland, pp.57-65.

[89] Torresen J. (2000), Possibilities and Limitations of Applying Evolvable Hardware to Real-World Applications, *Proc. of the 10th Int. Conf. on Field Programmable Logic and Applications*, Villach, Austria, pp. 230-239.

[90] Torresen J. (2000), Scalable evolvable hardware applied to road image recognition, *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, Palo Alto, CA, U.S.A, pp. 245-252.

[91] Tufte G. and Haddow P.C. (1999), Prototyping a GA Pipeline for Complete Hardware Evolution, *Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, Pasadena, CA, U.S.A, pp.18-25.

[92] Vassilev V., and Miller J.F. (2000), Embedding Landscape Neutrality To Build a Bridge from the Conventional to a More Efficient Three-bit Multiplier Circuit, *Proc. of the Genetic and Evolutionary Computation Conf.*, Las Vegas, NV, U.S.A.

[93] Vassilev V., and Miller J.F. (2000), The Advantages of Landscape Neutrality in Digital Circuit Evolution, *Proc. of the 3rd Int. Conf. on Evolvable Systems*, Edinburgh, U.K., pp. 252-263.

[94] Vassilev V., Miller J.F. and Fogarty T.C. (1999), On the Nature of Two-Bit Multiplier Landscapes, *Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, Pasadena, CA, U.S.A, pp.36-45.

[95] Yao X. and Higuchi T. (1996), Promises and Challenges of Evolvable Hardware, *Proc. of the 1st Int. Conf. on Evolvable Systems*, Tsukuba, Japan, pp. 55-78.

[96] Yih J.S. and Mazumder P. (1990), A Neural Network Design for Circuit Partitioning, *IEEE Trans. on Computer Aided Design*, **9**, no.10, pp. 1265-1271.

[97] Zebulum R.S., Aurélio Pacheo M. and Vellasco M. (1996), Evolvable Systems in Hardware Design: Taxonomy, Survey and Applications, *Proc. of the 1st Int. Conf. on Evolvable Systems*, Tsukuba, Japan, pp. 344-358.

[98] Zebulum R.S., Aurélio Pacheo M. and Vellasco M. (1997) Increasing Length Genotypes in Evolutionary Electronics, *Proc. of the 7th Int. Conf. on Genetic Algorithms*, East Lansing, MI, U.S.A.

[99] Zebulum R.S., Aurélio Pacheo M. and Vellasco M. (1998), Analog Circuits Evolution in Extrinsic and Intrinsic Modes, *Proc. of the 2nd Int. Conf. on Evolvable Systems*, Lausanne, Switzerland pp. 154-165.