Exploring Knowledge Schemes for Efficient Evolution of Hardware

Jim Torresen
Department of Informatics,
University of Oslo
P.O. Box 1080 Blindern
N-0316 Oslo, Norway
jimtoer@ifi.uio.no
http://www.ifi.uio.no/∼jimtoer

Abstract

There exist several approaches to improve the quality of evolution. In this paper, a priori design knowledge as a part of evolving systems is discussed. Further, experiments are reported showing how a priori knowledge (data buses and reuse) can be beneficial compared to gate level design of multiplier circuits. The future goal of the work is to be able to evolve systems for complex real-world applications (image and signal processing).

1. Introduction

Many would like evolvable systems to become crucial in the future development of computer systems as traditional design schemes are reaching their limits. Increasing size and complexity of electronic devices and systems have during the recent years led to a demand for new design schemes and tools. The new technology could be appropriate for systems implementing complex real-world applications within image and signal processing. Much research is now conducted on such applications to improve the performance as well as the speed of processing using traditional algorithms.

Evolvable hardware (EHW) is promising but there is still a long way to go before it is a real alternative to traditional design schemes [16]. By searching a larger design search space, EHW may find solutions for a task, unsolvable, or more optimal than those found using traditional design methods. However, this is also a problems since the search space easily becomes too large [10, 23]. Starting from scratch when evolving is not very biological motivated. Human beings try to apply all their earlier knowledge and skills when trying to solve a problem. If we are not able to solve it by ourselves, we search knowledge either in books/web etc. or by getting help from computer tools or other people, who have more knowledge or skills, to solve the problem. In evolution, there has been a tendency that introducing human expert knowledge would limit the exploration of the large search space. Thus, risking to loose interesting solutions that humans would not in their wildest dreams have thought of. This is true, but so far few such revolutionary systems have occurred.

Therefore, we find it appropriate to think it the other way around. If we tell the system what we know already, the evolution can go on from there to explore new and revolutionary systems. This is motivated by "inventing the wheel": If you know about the wheel, there is a much higher chance that you could invent a vehicle of some sort than if you did not. Thus, in this paper we would like to introduce some ideas about how human knowledge can be added to evolution. Further, we include some experimental results. For the rest of this introduction, we will give an overview of existing approaches applied to improve the evolvability of circuits and systems.

Various experiments on speeding up the GA computation have been undertaken [1]. The schemes involve fitness computation in parallel or a partitioned population evolved in parallel. Experiments are focussed on speeding up the GA computation, rather than dividing the application into subtasks. This approach assumes that GA finds a solution if it is allowed to compute enough generations. When small applications require a very long evolution time, there would probably be strict limitations on the systems evolvable even by parallel GA.

Other approaches to the problem have used variable length chromosomes [5]. Another option, called function level evolution, is to apply building blocks more complex than digital gates [12]. Most work is based on fixed functions. However, there has been work in Genetic Programming for *evolving* the functions [9]. The method is called Automatically Defined Functions (ADF). Experiments on extracting general principles when evolving systems of incremental size are reported in [11].

An improvement of artificial evolution — called co-evolution, has been proposed [4]. In co-evolution, a part of the data which defines the problem co-evolves simultaneously with a population of individuals solving the problem. This could lead to a solution with a better generalization than a solution based only on the initial data. A variant of co-evolution — called cooperative co-evolutionary algorithm, has been proposed by De Jong and Potter [6, 13]. It consists of parallel evolution of sub-structures, which interact to build more complex higher level structures. Complete solutions are obtained by assembling representatives from each group of sub-structures together. In that way, the fitness measure can be computed for the top level system. However, by testing a number of individuals from each sub-structure population, the fitness of individuals in a sub-population can be sorted according to their performance in the top-level system. Thus, no explicit local fitness measure for the sub-populations are applied in this approach. However, a mechanism is provided for initially seeding each GA population with user-supplied rules. Darwen and Yao have proposed a co-evolution scheme where the subpopulations are divided without human intervention [2]. There has been undertaken work on decomposition of logic functions by using evolution [7]. Results on evolving a 7-input and 10-output logic function show that such an approach is beneficial.

Incremental evolution for EHW was first introduced by Torresen in [14]. This is an approach that uses divide-and-conquer of the application. It has been shown that dividing the application is a very promising approach. It was proposed for EHW as a way to incrementally evolve a hardware system. The scheme is called *increased complexity evolution* since the system is evolved by evolving smaller sub-systems. Increased building block complexity is also a part of this approach, where the building blocks are becoming more complex as the system complexity increases.

Experiments show that the number of generations required for evolution by the new method can be substantially reduced compared to evolving a system directly in one operation. This has been shown both for a character recognition problem [15], a road image recognition problem [17] and prosthetic hand control [18]. In some experiments, better classification results than for artificial neural network were obtained. In addition, the hardware circuit was much smaller than what would have been required for running a neural network. Further, circuits for larger problems – than those evolvable by other schemes, have been evolved [21].

Through the research, the architecture as well as the incremental evolutionary algorithm have been extended and improved. It has been shown that even though the total number of generations is less, the performance of the evolved circuit is substantially better with the proposed scheme [18]. Thus, one achieves to solve complex problems where no good solution can be found by a single run evolution. Moreover, different ways of conducting the evolution have been proposed and tested. This includes evolving the best combination of circuits among a set of alternative circuits [20] and dynamic fitness functions in evolvable hardware [19]. Thus, we believe this approach is a good foundation for introducing *more* human *knowledge* into the design. That is, more manual design knowledge would be applied in the evolution. This is the topic for the work presented in this paper. By including more *a priori* knowledge, we should be able to evolve more complex and thus, more useful circuits than the circuits that have been evolvable so far.

X	*	**Y**	=	**Z**
00	*	00	=	0000
00	*	01	=	0000
00	*	10	=	0000
00	*	11	=	0000
01	*	00	=	0000
01	*	01	=	0001
01	*	10	=	0010
01	*	11	=	0011
10	*	00	=	0000
10	*	01	=	0010
10	*	10	=	0100
10	*	11	=	0110
11	*	00	=	0000
11	*	01	=	0011
11	*	10	=	0110
11	*	11	=	1001

Table 1. The truth table for multiplying two by two bits.

To make experiments, this paper reports about the evolution of multiplier circuits. Table 1 shows an example with the truth table for a two by two bit multiplier circuit. In the context of evolving multiplier circuits, the training set consists of all possible permutations of the inputs. Evolving this circuit in a single run requires a circuit with four inputs and four outputs.

The next section gives a listing of a priory knowledge available from traditional design. This is followed by a detailed description of an implemented scheme in Section 3. Results are reported in Section 4 and finally Section 5 gives the conclusions of the paper.

2 A priori Knowledge

This section gives an overview of different aspects of design knowledge. That is, we list those methods which a designer would apply to design a system. A priori hardware design knowledge is applied in a set of different ways:

- **Design specification.** Most traditional design is according to a predefined detailed *design specification* of the system. In evolution of digital circuits, normally only input-output vectors are specified. Specification of e.g. timing is rare even though this is often important in traditional design.

- **Partitioning the design task.** This regards how to best partition a given problem to solve. This design scheme is also called a hierarchical design scheme. For normal design, it will be a mixture of top-down and bottom-up design. A designer would normally start with preparing a top-level block description and continue by implementing one sub-circuit of the system at a time. He would usually know what is a reasonable partition of the design-task whereas this would normally not be available for a fully automatic design of a system. However, this would be quite similar to design based on increased complexity evolution presented in the introduction. We start with a complete data set that is partitioned (top-down). This is followed by a bottom-up evolution of the system. Thus, a priory knowledge is put into the top-down design only. Another aspect – at lower level, is to emphasize on the parts (or function) of the system not working. That is, when evolving a system, the fitness function should be adaptable to change its behavior according what is still left to be solved. This would have to be done in such a way that the parts working do not stop working.

- **Reuse.** Designing a large circuit would be almost impossible if the designer had to design each sub-circuit from scratch every time it is used [8]. Predesigned units for evolution could be simple Intellectual Property (IPs) cores, custom functional blocks or design library objects. Further, an interesting aspect is to reuse *evolved* structures several times in a system. This could either be about extracting promising parts of a chromosome [8] or apply evolved units in later evolution [14].

- **Data buses.** Almost no digital design is conducted without the use of data buses, while many systems are evolved with single bit wires only. To increase the complexity of a design, data buses would probably have to be included. Still, single bit lines will be needed. Thus, an evolvable architecture would have to consist of both busses and bit lines. This issue is also related to reuse with applying functional blocks in the evolution. A scheme for evolution with reuse and data buses is proposed in the next section.

- **Design optimization knowledge.** There exists a set of computer based optimization tools that could be applied in hardware design. That is, the designer specify a more or less abstract description of the system which the tool synthesize an optimized design for. Not much work has been published on combining evolution and optimization tools.

- **Prototyping.** This is very closely related to evolution. It consists of building various designs – with alternative architectures, to compare what is best. This is inherently what evolution consists of as well.

- **Hardware/software co-design.** Normally, hardware is developed in close cooperation with software being developed. What to implement in hardware and what to code as software are often an open question in the initial design phase. In evolution, designs are mainly either for software *or* hardware.

Not all of these issues are straightforward to combine with evolution. The one most explored (that probably will be more explored in the future) seems to be partitioning of the design task. Another one that probably could be useful is reuse. This could be at several levels. One of the problems with fixed length genetic algorithms is that a building block found in one part of the chromosome could not be reused in another part [8]. Further, standard crossover (with random crossover point) is not structure preserving. There have been introduced algorithms trying to overcome these problems. However, we feel that reuse should be further investigated when trying to build complex structures and systems. The next section describes work involving both reuse as well as data buses. Thus, this paper focus on a selected subset of a priori knowledge applied in evolution. In the future, the goal is to *combine* as much a priori knowledge as possible together to make advanced evolution.

3 Evolution with Data Buses

In this section, applying evolution on an architecture with data buses is explored. To demonstrate the scheme, we have selected a simple multiplier circuit. There has been work conducted earlier on evolving multipliers [22]. By applying increased complexity evolution it has been shown that large multiplier circuits can be evolved [21]. However, we are not aware of much work including data buses in digital circuit evolution. By applying this approach, we hope to improve the target architecture for evolution. Multiplier

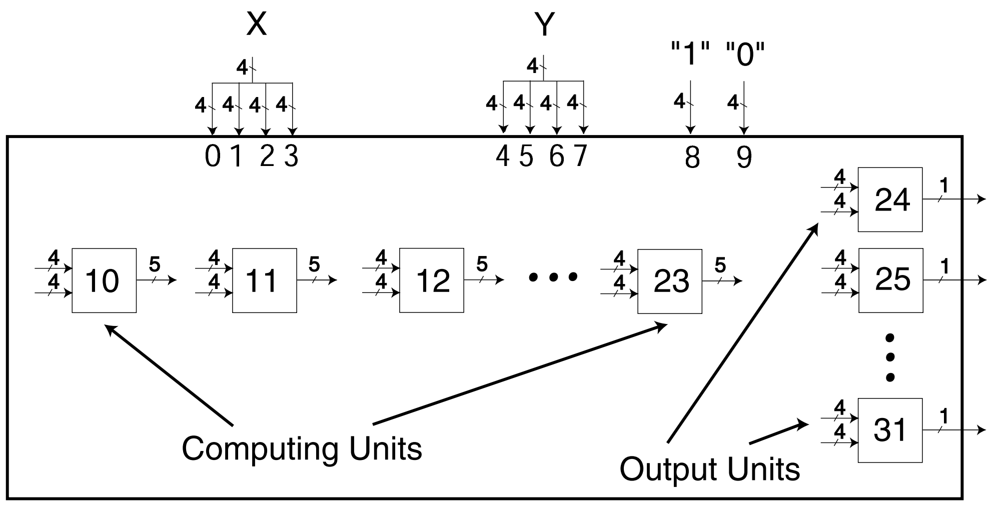

Figure 1. The evolvable architecture based on functional blocks and data buses.

circuits are appropriate since a standard design of a multiplier would consist of a set of parallel data buses together with AND gates and adder units (full adders).

E.g. for a 4 by 4 bits multiplier, the computation is as follows for multiplying $x_3x_2x_1x_0$ with $y_3y_2y_1y_0$:

$$\begin{array}{lllllll} & & & x_0y_3 & x_0y_2 & x_0y_1 & x_0y_0 \\ & & x_1y_3 & x_1y_2 & x_1y_1 & x_1y_0 & \\ & x_2y_3 & x_2y_2 & x_2y_1 & x_2y_0 & & \\ + x_3y_3 & x_3y_2 & x_3y_1 & x_3y_0 & & & \\ \hline \end{array}$$

Selecting appropriate building blocks for the task to solve is also a part of the *reuse* a priori knowledge. Thus, we apply the building blocks as shown in Figure 2.

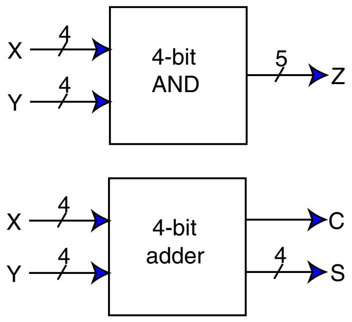

Figure 2. The building blocks used in evolution.

The function can be described as follows for the AND unit:

$$\begin{aligned} z_i &= x_iy_i \quad \text{where} \quad i = 1,2,3,4 \\ z_5 &= 0 \end{aligned}$$

The adder is given by:

$$\left.\begin{aligned} s_i &= (x_i + y_i + c_{i-1}) \bmod 2 \\ c_{i+1} &= \tfrac{(x_i+y_i+c_{i-1})}{2} \\ c &= c_5 \end{aligned}\right\} \text{where } i = 1,2,3,4$$

$$c_0 = 0$$

Thus, these functional blocks allow us to evolve with 4 bits buses. The architecture, that will be applied for evolving multipliers, is shown in Figure 1. It consists of a number of functional units (given in Figure 2) for computation and output, respectively. A unit can connect to any unit with a lower *id* than itself including the inputs. In addition to inputing the two four bits numbers, constants with “1” and “0” are input as well. The numbers input are duplicated four times to make the probability higher for connecting to input numbers rather than other units. This is since a normal design would have such a higher connectivity to the input numbers. One output unit is assigned to *each* output bit. However, in the future it should be considered if multi-bit output units are beneficial.

Ordinary designs can not be based on data buses only. There would also have to be a possibility for single bit lines in the design. The following connections between units are possible here:

- Four output bits connected to four input bits.
- A single bit source connected to four inputs.

As seen in Figure 3, during evolution it is allowed to connect either to a 4 bits bus output (A to C in Figure 3) or *one* of the lines in the bus (B to C in Figure 3). This corresponds well with the multiplication operation as seen above. Which

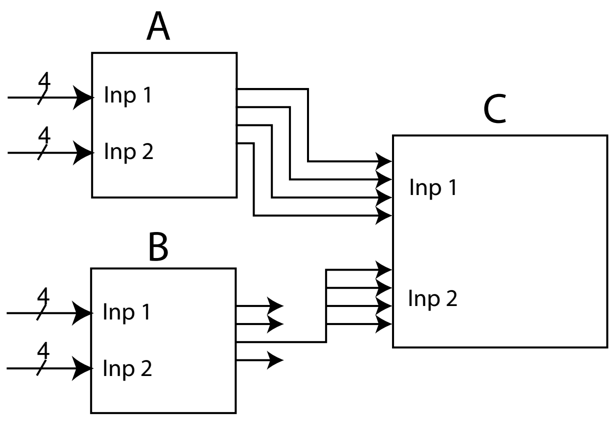

Figure 3. The two types of connection between units: Data bus (A to C) and single bit line (B to C).

type of connection to apply for each input is given by bits in the chromosome string. Further, which single bit line to select is determined by bits in the chromosome as well.

The adder unit needs *five* output bits. Thus, those units connecting to the output of such a unit must select either connecting to the upper or lower four bits. However, to allow for shift operation in the architecture, the *same* two connections are also possible to the AND unit. The most significant output bit (z_5) of each AND unit is set to “0”. The complete chromosome coding for one unit is given in Table 2. The numbers in parenthesis are the number of bits used. In total, 19 bits are applied and this results in a chromosome (32-10) x 19 bits = 418 bits long. It is actually slightly shorter since a unit with a small *id* needs less than 5 bits to address units with a smaller *id* than itself. Evolving with 4 bits functional blocks provides a more compact chromosome string than building blocks consisting of gates. This is due to more complexity is provided by each building block. Such an architecture allows for connection to *all* previous units. This would normally not be possible in a gate based architecture since this would make the chromosome string very long.

By looking at the multiplication operation, we extracted and applied some more a priori knowledge. When a unit connecting to *input numbers only*, the following applies:

- Force one of the two inputs to be bus and the other to be bit line. For a traditional design of a multiplier (with two input numbers) there are always a single bit and a bus together.
- A unit is made to always connect its two inputs to the two *different* numbers. Thus, it would not be possible for a unit connecting to the same number twice.
- Forcing the unit to be AND since bits in the input numbers first have to be multiplied (AND operation) before addition can take place.

3.1 A Gate Based Architecture

To be able to compare the results with gate level evolution, we have also run experiments with a gate array based architecture (similar to the one applied in [21]). The target hardware consists of a fixed-size array of logic gates. The array consists of *n* gate layers from input to output as seen in Figure 4. Each logic gate (LG) is either *Buffer*, *AND*, *XOR* or *MUX* gate. Each gate’s three inputs in layer l is connected to the outputs of three gates in layer $l-1$. In the following experiments, the array consists of 8 layers each consisting of 16 gates. A 4$\times$4-bit multiplier consists of 8 inputs and 8 outputs. Inverted versions of the inputs are input as well.

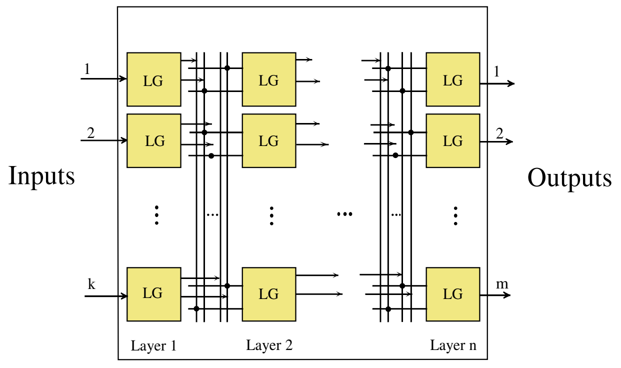

Figure 4. The architecture of the gate array.

The *function* of each gate and its *three inputs* are determined by evolution. The encoding of each logic gate in the chromosome string is as follows:

Input 1 (4)	Input 2 (4)	Input 3 (4)	Function (2)

For the given array, the chromosome string length becomes 1792 bits long.

3.2 GA Parameters and Fitness Function

Various experiments were undertaken to find appropriate GA parameters. The ones that seemed to give the best results were selected and fixed for all the experiments. This was necessary due to the large number of experiments that would have been required if GA parameters should be able vary through all the experiments. The preliminary experiments indicated that the parameter setting was not a major critical issue.

The simple GA style – given by Goldberg [3], was applied for the evolution with a population size of 50. For each new generation an entirely new population of individuals is generated. Elitism is used, thus, the best individuals

Function (1)	Input 1/2 (2x5)	Bus or bit (2x1)	Which bit in bus (2x2)	High or low outp (2x1)

Table 2. The chromosome representation for one unit in the architecture in Figure 1.

from each generation are carried over to the next generation. The (single point) crossover rate is 0.8 (0.5 for the gate array), thus the cloning rate is 0.2. Roulette wheel selection scheme is applied. The mutation rate – the probability of bit inversion for each bit in the binary chromosome string, is 0.01 (0.005 for the gate array). Each experiment has been run five times, each for 50,000 generations.

The fitness function is computed in the following way:

$$F = \sum_{\text{vec}} \sum_{\text{outp}} x \qquad \text{where } x = \begin{cases} 0 & \text{if } y \neq d \\ 1 & \text{if } y = d = 0 \\ 3 & \text{if } y = d = 1 \end{cases}$$

For each output, the computed output y is compared to the target d. If these equal and the value is equal to zero then 1 is added to the fitness function. On the other hand, if they equal and the value is equal to one, 3 is added. In this way, an emphasize is given to the outputs being one (which is less number than those being 0). This has shown to be important for getting faster evolution of well performing circuits. The function sum these values for the outputs (outp) of all the truth table vectors (vec).

The proposed architecture fits into most FPGAs (Field Programmable Gate Arrays). The evolution is undertaken off-line using software simulation. However, since no feedback connections are used and the number of gates between the input and output is limited, the real performance should equal the simulation. Any spikes could be removed using registers in the circuit. The number of gates used in the architectures is not considered since todays FPGAs contain a large number of gates.

4 Results

In this section, results from a set of experiments are included. Three different multipliers have been evolved: 2x4-bit, 3x4-bit and 4x4-bit. They were evolved both with a traditional gate array (as described in Section 3.1) and the new proposed data bus architecture (first part of Section 3). For every multiplier, the results were best when applying the data bus based architecture, as seen in Figure 5. The performance is given as percentage of the maximum fitness.

Table 3 summarizes the main results. The average fitness is 5-7% higher for the data bus architecture. Further, the *minimum* value for the data bus architecture is always larger than the *maximum* value for the gate array architecture. As can be seen however, it was not possible to evolve correctly working multiplier circuits. This is both due to

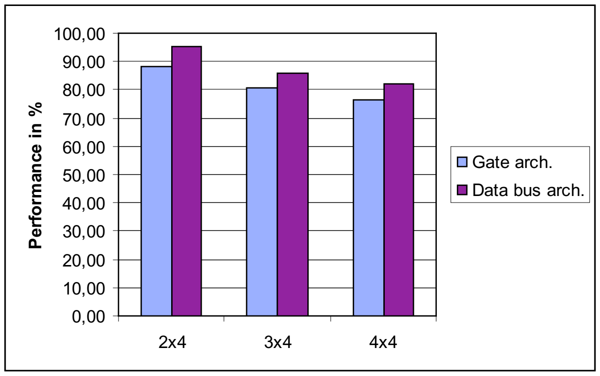

Figure 5. Comparing the average fitness for evolving three different multiplier circuits.

the complexity of the problem as well as the fact that the experiments that have been run are limited.

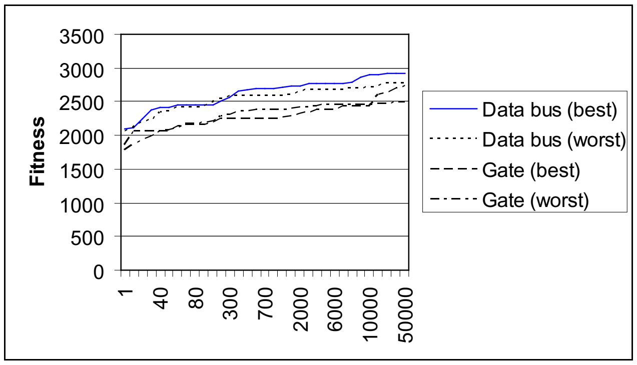

Figure 6. The fitness of the best and worst circuit (out of five runs) throughout evolution of a 4x4-bit multiplier circuit. The x-axis is the number of generations.

Figure 6 plots the fitness throughout evolution of the best and worst performing circuit (each selected among five runs) of the 4x4-bit multiplier. All the time, the data bus based architecture performs better than the gate array.

Even though it was not possible to obtain a fully working multiplier circuit, we regard the results as an important step in the right direction. The results indicate an improvement compared to evolving with gates as building blocks. If more time had been available, to run the experiments for a larger

Multiplier	Type of arch.	# vectors	# outputs	% of max fitness		
				Min	Max	Avr
2x4 bit	Gate	64	6	87.4	89.4	88.0
2x4 bit	Data bus	64	6	94.3	96.5	95.4
3x4 bit	Gate	128	7	78.9	82.1	80.7
3x4 bit	Data bus	128	7	84.8	87.3	85.7
4x4 bit	Gate	256	8	72.8	79.1	76.4
4x4 bit	Data bus	256	8	79.8	84.7	82.1

Table 3. The performance of multiplier circuits of size 2x4-bit, 3x4-bit and 4x4-bit.

number of generations, it is expected that the performance could have been further improved. Moreover, by combining the scheme proposed in this paper with earlier methods like incremental evolution, advanced systems should be evolvable. This a part the future work.

5 Conclusions

In this paper, a number of traditional design aspects have been discussed. Further, experiments involving *reuse* and *data buses* in evolution of multiplier circuits have been reported. Experiments are promising and by further developing the architecture and the a priory knowledge inclusion, systems for complex real-world applications should be evolvable.

Acknowledgments

The research presented in this paper is a part of the project *Biological-Inspired Design of Systems for Complex Real-World Applications* (project number 160308/V30) funded by the The Research Council of Norway.

References

[1] E. Cantu-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.

[2] P. Darwen and X. Yao. Automatic modularization by speciation. In *Proc. of 1996 IEEE International Conference on Evolutionary Computation*, pages 88–93, 1996.

[3] D. Goldberg. *Genetic Algorithms in search, optimization, and machine learning*. Addison–Wesley, 1989.

[4] W. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1-3):228–234, 1990.

[5] M. Iwata, I. Kajitani, H. Yamada, H. Iba, and T. Higuchi. A pattern recognition system using evolvable hardware. In *Proc. of Parallel Problem Solving from Nature IV (PPSN IV)*, volume 1141 of *Lecture Notes in Computer Science*, pages 761–770. Springer-Verlag, September 1996.

[6] K. D. Jong and M. Potter. Evolving complex structures via co-operative coevolution. In *Proc. of Fourth Annual Conference on Evolutionary Programming*, pages 307–317. MIT Press, 1995.

[7] T. Kalganova. Bidirectional incremental evolution in extrinsic evolvable hardware. In J. L. et al., editor, *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 65–74. IEEE Computer Society, Silicon Valley, USA, July 2000.

[8] J. Koza et al. The importance of reuse and development in evolvable hardware. In J. L. et al., editor, *Proc. of the 2003 NASA/DoD Conference on Evolvable Hardware*, pages 33–42. IEEE, 2003.

[9] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press, 1994.

[10] W.-P. Lee, J. Hallam, and H. Lund. Learning complex robot behaviours by evolutionary computing with task decomposition. In A. Birk and J. Demiris, editors, *Learning Robots: Proc. of 6th European Workshop, EWLR-6 Brighton*, volume 1545 of *Lecture Notes in Artificial Intelligence*, pages 155–172. Springer-Verlag, 1997.

[11] J. Miller, D. Job, and V. Vassilev. Principles in the evolutionary design of digital circuits – Part I. *Journal of Genetic Programming and Evolvable Machines*, 1(1):8–35, 2000.

[12] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. Hardware evolution at function level. In *Proc. of Parallel Problem Solving from Nature IV (PPSN IV)*, volume 1141 of *Lecture Notes in Computer Science*, pages 62–71. Springer-Verlag, September 1996.

[13] M. Potter and K. D. Jong. Evolving neural networks with collaborative species. In *Proc. of Summer Computer Simulation Conference*. The Society for Computer Simulation, 1995.

[14] J. Torresen. A divide-and-conquer approach to evolvable hardware. In M. Sipper et al., editors, *Evolvable Systems: From Biology to Hardware. Second International Conference, ICES 98*, volume 1478 of *Lecture Notes in Computer Science*, pages 57–65. Springer-Verlag, 1998.

[15] J. Torresen. Increased complexity evolution applied to evolvable hardware. In Dagli et al., editors, *Smart Engineering System Design: Neural Networks, Fuzzy Logic , Evolutionary Programming, Data Mining, and Complex Systems, Proc. of ANNIE'99*, pages 429–436. ASME Press, November 1999.

[16] J. Torresen. Possibilities and limitations of applying evolvable hardware to real-world application. In R. Hartenstein et al., editors, *Field-Programmable Logic and Applications: 10th International Conference on Field Programmable Logic and Applications (FPL-2000)*, volume 1896 of *Lecture Notes in Computer Science*, pages 230–239. Springer-Verlag, 2000.

[17] J. Torresen. Scalable evolvable hardware applied to road image recognition. In J. L. et al., editor, *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 245–252. IEEE Computer Society, Silicon Valley, USA, July 2000.

[18] J. Torresen. Two-step incremental evolution of a digital logic gate based prosthetic hand controller. In *Evolvable Systems: From Biology to Hardware. Fourth International Conference, (ICES'01)*, volume 2210 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2001.

[19] J. Torresen. A dynamic fitness function applied to improve the generalisation when evolving a signal processing hardware architecture. In *Applications of Evolutionary Computing: EvoWorkshops 2002*, volume 2279 of *Lecture Notes in Computer Science*, pages 267–279. Springer-Verlag, 2002.

[20] J. Torresen. Evolving both hardware subsystems and the selection of variants of such into an assembled system. In *Proc. of the 16th European Simulation Multiconference (ESM2002)*, pages 451–457. SCS Europe, June 2002.

[21] J. Torresen. Evolving multiplier circuits by training set and training vector partitioning. In P. H. A. Tyrrel and J. Torresen, editors, *Evolvable Systems: From Biology to Hardware. Fifth International Conference, ICES'03*, volume 2606 of *Lecture Notes in Computer Science*, pages 228–237. Springer-Verlag, 2003.

[22] D. J. V. Vassilev and J. Miller. Towards the automatic design of more efficient digital circuits. In J. L. et al., editor, *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 151–160. IEEE Computer Society, Silicon Valley, USA, July 2000.

[23] X. Yao and T. Higuchi. Promises and challenges of evolvable hardware. In T. Higuchi et al., editors, *Evolvable Systems: From Biology to Hardware. First International Conference, ICES 96*, volume 1259 of *Lecture Notes in Computer Science*, pages 55–78. Springer-Verlag, 1997.