A Pipelined Hardware implementation of Genetic Programming using FPGAs and Handel-C

Peter Martin

Department of Computer Science, University of Essex,
Wivenhoe Park, Colchester, CO4 3SQ, UK.

Abstract A complete Genetic Programming (GP) system implemented in a single FPGA is described in this paper. The GP system is capable of solving problems that require large populations and by using parallel fitness evaluations can solve problems in a much shorter time that a conventional GP system in software. A high level language to hardware compilation system called Handel-C is used for implementation.

1 Introduction

The motivation for this work is that as problems get harder, the performance of traditional computers can be severely stretched despite the continuing increase in performance of modern CPUs. By implementing a GP system directly in hardware the aim is to increase the performance by a sufficiently large factor to be able to tackle harder problems and to make investigations into the operation of GP easier. This paper is an update to the work presented in [9] which describes how a GP system that includes initial population creation, fitness evaluation, selection, and breeding operators can be implemented in a Field Programmable Gate Array (FPGA) using a high level language to hardware compilation technique. Two major areas were singled out for further work in order to improve the performance: 1) extend the implementation to handle larger populations; 2) the use of pipelining to improve the parallelism of the hardware.

The changes needed and the results of implementing the changes are described in this paper. The paper begins with a brief survey of previous work using FPGAs for evolutionary techniques and a short summary of the Handel-C language and the target hardware. This is followed by a description of the revised design that stores the population in off-chip Static Random Access Memory (SRAM) and that also uses pipelining. The experimental setup is presented together with some results that illustrate the effect of the changes. The changes are then discussed and areas for further work are suggested.

2 Previous work using FPGAs in Evolutionary Computing

FPGAs have featured in the field of evolutionary computing under three distinct headings:

1) as a means of implementing the fitness functions of Genetic Algorithms or Genetic Programming [6,17];

2) as a platform for implementing a Genetic or Evolutionary Algorithm [3,4,12,13,14];

3) in relation to evolving hardware by means of an evolutionary technique [2,8,15,16].

A more detailed review of this and other work can be found in [9].

3 Description of Handel-C and the target hardware

Handel-C is a high level language that is at the heart of a hardware compilation system known as Celoxica DK1 [1] which is designed to compile programs written in a C-like high level language into synchronous hardware. Since Handel-C targets hardware, there are some programming restrictions when compared to using ISO-C, and these need to be considered when designing code that can be compiled by Handel-C. Some of these restrictions particularly affect the building of a GP system. Firstly, there is no stack available, so recursive functions cannot be directly supported by the language. Secondly, there is a severe limit to the size of memory that can be implemented using standard logic cells on an FPGA because implementing memory is expensive in terms of silicon real estate. However, some FPGAs have internal RAM that can be used by Handel-C which is supported by the `ram` storage specifier.

The target hardware for this work is a Celoxica RC1000 FPGA development board fitted with a Xilinx XCV2000E Virtex-E FPGA having 43,200 logic cells and 655,360 bits of block ram, a PCI bridge that communicates between the RC1000 board and the host computer's PCI bus, and four banks of Static Random Access Memory (SRAM). Logic circuits isolate the FPGA from the SRAM, allowing both the host CPU and the FPGA to access the SRAM, though not concurrently.

4 System Architecture

The lack of a stack in Handel-C means that a standard tree based representation is difficult to implement because recursion cannot be handled by the language. Instead, a linear program representation is used [11], though other compact representations such as Cartesian Genetic Programming [10] in which programs are represented as graphs, are also worth considering. Using a linear representation, a program consists of a sequence of words which are interpreted by the problem specific fitness function. To ease the design, each program has a fixed maximum length that it can grow to. Crossover is performed by selecting crossover points at random in two individuals and swapping the nodes after the crossover points. If the length of a program would exceed the maximum, it is simply truncated to the maximum. Mutation is performed on an individual by replacing a word with a new randomly generated word which has the potential effect of changing

both the functionality and the terminals of that node. The operators are chosen using an 8 bit random number and bit masks which eliminates less-than or greater-than comparisons which are inefficient in terms of logic. The probability of selecting each operator is 31/255 (12%) for mutation, 63/255 (24.5%) for copy and the remainder (63.5%) for crossover.

4.1 Extending the population size

Large populations are supported by storing the entire population in off-chip SRAM. The Celoxica RC1000 board has 8 MiB[1] of SRAM arranged as 4 banks of 2 MiB that can be directly addressed by the FPGA, and each bank is configured as 512 Ki 32bit words. In practice, one bank is reserved for storing the results of the run (fitness and lengths of each individual), leaving three banks available for the population. The total population size is determined by the program size chosen and the size of the program nodes. Table 1 illustrates the potential range that can be accommodated for a node size of 32 bits.

Table 1. Possible population sizes when using three 2 MiB memory banks and a word size of 32 bits for different program sizes.

Max. Program Length (words)	16	32	64	128	256	512	1024
Max. population size	98,304	49,152	24,576	12,288	6,144	3,072	1,536

External SRAM can only be written to or read from once per clock cycle, so care was taken in the design to ensure that parallel access to memory cannot occur. Similarly, the on-chip block select RAM must not be accessed more than once per clock cycle. Concurrent access to the block select RAMs is achieved by partitioning the rams into smaller blocks that can be accessed in parallel. Access to the SRAM is controlled by the pipeline.

4.2 Using pipelines to improve performance

Implementing algorithmic parallelism or pipelining is a frequently used technique in hardware design that reduces the number of clock cycles needed to perform complex operations. Pipelines can be implemented at a number of levels and in this work pipelines have been used in several places; a high-level coarse grained control pipeline, and fine grained pipelines in the fitness evaluation function and functions that copy data to/from SRAM. The four major GP operations are divided among the stages of the pipeline: selection of individuals from the population for breeding, breeding new individuals, fitness evaluation of the individuals, and replacement of the new individuals in the population. Because of the need to control access to the main population in SRAM during the selection phase which reads individuals from SRAM into block ram, and writing modified programs back to SRAM, these two operations are combined into one stage. This

[1] This paper uses the IEC recommended prefixes for binary multiples. MiB indicates 2^{20} bytes.

leaves the breeding and fitness evaluation/replacement operations. Breeding is closely tied to selection and needs to occur before evaluation can take place so this is combined into the selection phase. Figure 1 illustrates the resultant architecture and the coarse-grained control pipeline. Stage1 is a pseudo random

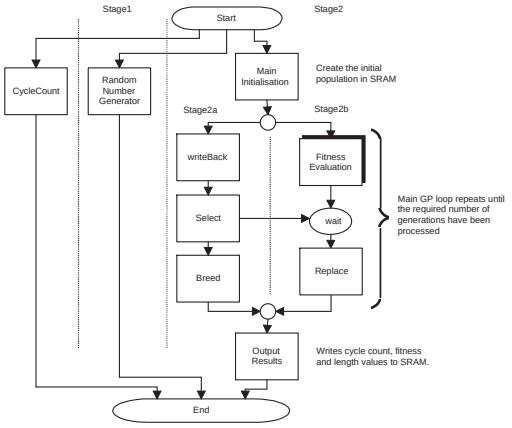

Figure 1. Overall architecture of the pipelined GP system.

number generator based on a logical feedback shift register (LFSR), which runs continuously, generating a new random number every clock cycle. The random numbers are available to the rest of the machine with no overhead. Stage2 is the main GP machine and consists of two sub-stages. Communication between the WriteBack/Select/Breed sub-stage (stage 2a) and the Evaluate/Replace sub-stage (stage 2b) is via a two dimensional array of individuals in block ram, indexed by a global phase index which is toggled each time stages 2a and 2b complete. The WriteBack phase updates the main population in SRAM with the result of the preceding Evaluate/Replace phase. The Select phase selects a series of two parents using tournament selection and copies the selected individuals from SRAM to the on-chip working RAM and the Breed phase then creates a series of new individuals ready for stage 2b to use. Stage 2b performs parallel evaluations of the individuals and then determines which individuals should be replaced. Parallel evaluation is achieved by replicating the hardware for fitness evaluation. The individuals identified for replacement are in turn written back to the main population at the start of the next WriteBack/Select/Breed sub-stage. The wait between evaluation and replacement is needed because both selection and replacement require access to the global fitness vector. In practice this only comes into play when the evaluation phase takes less time than WriteBack and Selection, which only happens for very simple fitness functions.

A finer grained level of pipelining is implemented in the fitness evaluation function. FPGAs are synchronous devices, meaning that a clock is used to latch data into registers. In Handel-C all expressions are implemented using combinatorial logic, which if allowed to grow in depth can restrict the maximum frequency

the FPGA can be clocked at. This is because of the delays introduced by the combinatorial paths. Therefore, to reduce logic depth, and hence improve on the clock frequency, it is often advantageous to split a complex expression into more but simpler expressions. This usually requires more clock cycles, but by pipelining the operations an effective single cycle throughput can be achieved. In this design, the function read, and decode is pipelined with the function evaluation, though the effectiveness of this is problem specific.

In conventional steady-state GP, once an individual has been evaluated it replaces the worst individual in the population. In a hardware implementation with parallel fitness evaluations this is expensive to implement since a global search is required. An alternative to this called survival-driven evolution, has been successfully used by Shackleford et al [14]. In this scheme only offspring that are fitter than the worst of their two parents will survive into the next generation by replacing one of the parents. This removes the need for any global search and this scheme was adapted to the current work by maintaining a record of the parents of each individual.

To compare the performance of different implementations a way of measuring the number of cycles used by the FPGA is needed. One possibility is to use the DK1 simulator, but in large designs with long running times this can take many hours of running which is often impractical. An alternative is to include a cycle counter in the design which can be read by external programs. The internal cycle counter runs in parallel with the rest of the hardware, incrementing a counter once per clock cycle. This approach could be extended to providing fine-grained measurement of the cycles required by the individual phases which would be valuable for evaluating the detailed performance of the design.

Once the GP machine has finished a run, the best program needs to be communicated to the outside world. The individual programs are already in SRAM, so they can be read directly by the host. The program fitness and lengths are written to SRAM when the GP machine has finished so they can also be read by the host. In addition, the cycle count(s), and other parameters are made available to the host via SRAM.

5 Experimental setup

To evaluate the effect of the changes made, two experiments were performed. Firstly, a direct comparison with the previous design using the XOR problem was run. This was done to gauge the overall effect of storing the population in off-chip SRAM, and of implementing the pipelines. Next the Santa Fe Ant problem was implemented both as a demonstration of a hard problem and an example of where the function set is equivalent to that of a traditional CPU instruction set.

Each experiment was run using four different environments. Firstly, an ISO-C version of the algorithm was developed to prove the operation of the program because debugging tools for standard C are readily available, and the development time for C is considerably shorter than when using Handel-C and FPGA

tools. Secondly, a PowerPC simulator was used to measure how many cycles the algorithm needed when run on a typical Reduced Instruction Set Computer (RISC) and thirdly, the design was implemented on an FPGA. Each problem was implemented with a range of parallel evaluations. Lastly to get a feel for the total effect of implementing a GP system in hardware when compared to using a popular software GP system, the problems were implemented using lilgp and a comparison made of the performance.

When the design is implemented on an FPGA, a host control program configures the FPGA, sets up the random number seed and other control parameters in SRAM, and initiates the GP run. Once the GP run has finished, the FPGA signals the host program and the host control program reads the fitness values from SRAM, decides which program(s) are suitable, reads the appropriate SRAM locations and outputs the program in a usable form. For the two experiments, host programs were also written to verify the programs, and in the case of the ant problem, display a simple graphical trace of the ant's behavior.

6 Experiment Descriptions and Results

6.1 Exclusive Or Problem

Description The 2 bit XOR function $x = (a\overline{b}) + (\overline{a}b)$ uses the four basic two input logic primitives AND, OR, NOR and NAND which take two registers, R_a and R_b. The result is placed into R_a. These functions have been shown to be sufficient to solve the boolean XOR problem [5]. Execution is terminated when the last instruction in the program has been executed. The two inputs a and b were written to registers R_0 and R_1 before the fitness evaluation, and the result x read from register R_0 after the fitness evaluation. The full set of parameters is given in Table 2.

Table 2. Parameters for the XOR problem

Parameter	Value
Population Size	16
Functions	AND(R_a,R_b), OR(R_a,R_b), NOR(R_a,R_b), NAND(R_a,R_b)
Terminals	4 registers
Max Program Size	16
Generations	511
Fitness Cases	4 pairs of values of a and b
Raw Fitness	The number of fitness cases that failed to yield the expected result.

Comparing these results first with the results in [9] which achieved a $Speedup_{time}$ of 6 times for 4 parallel evaluations, it can be seen that splitting the algorithm into two sub-stages gives a useful increase in performance. However, the surprising result is that it takes longer to run the XOR problem when more evaluations are performed in parallel, in particular when 8 parallel evaluations are done. Detailed investigation showed that this was a side effect of the selection method.

Table 3. Results of running the XOR problem. The results are the average of 10 runs for each configuration, each run using a different random seed.

Measurement	PowerPC	HandelC			
Parallel fitness evaluations	n/a	1	2	4	8
Cycles	13,723,187	74,819	73,232	72,184	81,767
Clock Frequency	200MHz	52MHz	48MHz	42MHz	37MHz
Number of Slices	n/a	1238	1247	1725	2801
$Speedup_{cycles}$	1	183	187	190	167
$Speedup_{time}$	1	47	44	39	31

During selection the number of individuals selected from the main population is the number of parallel fitness evaluations wanted, and these are selected at random from the population, but only those individuals that are not currently being evaluated by the Evaluate/Replace sub-stage are valid candidates. When the number of individuals required is half the population size, many more attempts must be made by the selection phase to find valid individuals. This explains why when the number of parallel evaluations is 8, the run time is greater than when only two individuals are being selected.

The frequencies in table 3 for the Handel-C implementations is that reported by the place&route tools, and takes into account the delays introduced by the combinatorial logic and the delays introduced by the routing resources used on the FPGA. A lot of effort was spent to reduce the logic and routing delays in the design, with the result that this design runs substantially faster that the previous design which could only reach 18 MHz.

Running this problem with lilgp required approximately 165×10^9 cycles, or more than 12 times the number of cycles needed by the linear implementation.

6.2 Artificial Ant Problem

Description The motivation for choosing this problem for a hardware implementation is two fold: Firstly it is a hard problem for GP to solve [7], and secondly it demonstrates that a custom hardware design can efficiently encode the function and terminal set as native 'instructions'. That is to say one of the attractions of using an FPGA is that custom instructions not normally found in production CPUs can easily be constructed. The full set of parameters is given in Table 4. The ANT problem was executed using the same environments as the XOR problem and the results are presented in Table 5.

An example program from this problem found in one run is:

```
IF_FOOD(LEFT,RIGHT)
PROGN (NOP,RIGHT)
IF_FOOD (NOP,LEFT)
PROGN (MOVE,LEFT)
```

Figure 2 shows the speedup results for the Ant problem, and gives both the $Speedup_{cycles}$ and $Speedup_{time}$. These results show that for the Ant problem, increasing the number of parallel fitness evaluations increases the $Speedup_{cycles}$

Table 4. Parameters for the ANT problem

Parameter	Value
Population Size	512
Functions	IF_FOOD(Ta,T_b), PROGN(Ta, T_b)
Terminals	MOVE, LEFT, RIGHT, NOP
Max Program Size	32
Generations	511
Fitness Cases	One fitness case. The program was run until 1024 timesteps had elapsed or the ant had consumed all the food.
Raw Fitness	The number of pieces of food not eaten in the time available.

Table 5. Results of running the ANT problem

Measurement	PowerPC	HandelC				
Parallel fitness evaluations	n/a	2	4	8	16	32
Cycles	2.695e9	42.58e6	23.19e6	13.15e6	7.53e6	4.2e6
Clock Frequency	200MHz	40	38	36	33	31
Number of Slices	n/a	1,835	2,636	4,840	7,429	14,908
$Speedup_{cycles}$	1	69	116	204	358	642
$Speedup_{time}$	1	13	22	36	59	99

factor linearly, but because the routing delay on the FPGA increases with larger designs, the maximum clock frequency decreases, offsetting some of the gains made by increasing the parallelism. The number of slices used for 32 parallel

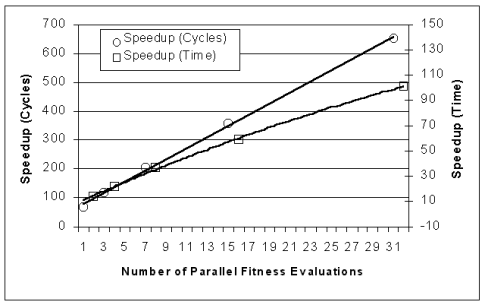

Figure 2. The speedup factors for the number of cycles ($Speedup_{cycles}$) and the time ($Speedup_{time}$) for the Ant problem.

evaluations is nearly 80% of the total available on the chip. This is effectively the limit for the XCV2000E FPGA. It is worth remarking that a PC with 750 MiB of RAM was required to run the Handel-C compiler and the place&route tools before this design could be implemented, and a PC with a 1.4G Hz Athlon CPU required nearly four hours to complete the place&route. This is in contrast to a modest 500 MHz Pentium machine capable of compiling and running lilgp and other popular GP packages.

Running this problem using lilgp in the PPC simulator needed approximately 8.6×10^9 cycles, or over three times the number required by the linear implemen-

tation on a PPC processor, giving the FPGA using 32 parallel fitness evaluations a $Speedup_{cycles}$ of 2047 and a $Speedup_{time}$ of approximately 300 times over lilgp.

7 Discussion

7.1 Effect of Implementing Pipelines and increasing parallelism

A direct comparison between the work in [9] showed for the XOR problem that using a pipeline in the main control loop, and in the fitness function provides a useful speedup, but because the fitness evaluation is much shorter than the WriteBack/Select/Breed sub-stage, there is no benefit to increasing the number of parallel fitness evaluations. It was also clear that for small populations there is a limit to the number of parallel fitness evaluations that can be accommodated. However, the situation is reversed in the Ant problem because the time needed for fitness evaluation is much larger than the WriteBack/Select/Breed phase. Clearly for other problems where the fitness function takes a long time it would be worth devising efficient pipelines for the fitness evaluation functions.

7.2 Comparison to a popular Software GP system.

While this may appear to be an unfair comparison, a lot of work in GP is centered around exploring the detailed operation of GP, which often requires hundreds or thousands of runs with minor parameter changes, and performance is still likely to be an issue even with processors running at 2 GHz and beyond. This comparison was done to see if using a hardware implementation would be of benefit to researchers. The results show that where a fixed problem type needs to be run many times, a hardware implementation using many parallel fitness evaluations could reduce the time required for extended runs by over two orders of magnitude. While this looks promising, it must also be noted that changing parameters for a FPGA run requires a large investment in time and the hardware approach may not always be suitable.

7.3 Handel-C as an Implementation Language

Using Handel-C to implement this design has highlighted two major benefits: Firstly, the design of the system by someone trained as a software engineer with limited hardware experience was relatively straightforward. Secondly, it meant that the algorithm could be tested and debugged using traditional software tools. This is important when the time to compile a Handel-C design for the larger problems into a simulation on a 1.4 GHz PC is of the order of 25 minutes, and to place and route could take an hour or more. In contrast, compilation of the code using the GNU compiler took seconds.

An implication of using Handel-C is that to get the best throughput some familiar programming constructs must be abandoned and new techniques adopted. It is not possible to completely ignore the hardware aspects of a design, for example when interfacing to SRAM, and when trying to squeeze the last few nano seconds out of the design during the final place and route stages.

8 Further Work

The latest Virtex-II FPGAs from Xilinx are bigger and faster than the Virtex-E device used so far. These devices promise even better speedups, and initial studies using these devices have indicated that a further speedup of two to three times is possible. The Virtex-II devices also have larger gate counts and bigger on-chip block RAMs. The largest of these devices - the XC2V8000 could theoretically support up to 128 parallel fitness evaluations for the artificial ant problem. The initial indication is that by combining the faster clock frequency and more parallel fitness evaluations the ant problem could see a further 10 times speedup. Further work needs to be done to verify this.

Whilst the performance gains are substantial for one of the problems looked at, the costs associated with using this technology need to be evaluated with respect to software engineering effort, capital equipment costs and operational considerations, so that practitioners can make the appropriate choices when choosing a technology for implementing a GP system.

9 Conclusions

Moving the population storage to off-chip SRAM has allowed the design to solve problems requiring larger population sizes, an example of which is the Ant problem. Pipelining the breed, selection and evaluation phases gives a performance boost to problems that have short evaluation time requirements like the XOR problem. For problems like the Ant problem that require extended fitness evaluation times the benefits of using a pipeline are even greater, allowing the number of parallel fitness evaluations to be increased and the performance increasing in a nearly linear relationship.

References

1. Celoxica. Web site of Celoxica Ltd. www.celoxica.com, 2001. Vendors of Handel-C. Last visited 15/June/2001.
2. T. Fogarty, J. Miller, and P. Thompson. Evolving Digital Logic Circuits on Xilinx 6000 Family FPGAs. In P. Chawdhry, R. Roy, and R. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 299–305. Springer-Verlag, 1998.
3. P. Graham and B. Nelson. Genetic Algorithms In Software and In hardware - A Performance Analysis Of Workstation and Custom Computing Machine Implementations. In K. Pocek and J. Arnold, editors, *Proceedings of the Fourth IEEE Symposium of FPGAs for Custom Computing Machines.*, pages 216–225, Napa Valley, Califormia, Apr. 1996. IEEE Computer Society Press.
4. M. Heywood and A. Zincir-Heywood. Register based genetic programming on FPGA computing platforms. In R. Poli, W. Banzhaf, W. Langdon, J. Miller, P. Nordin, and T. Fogarty, editors, *Genetic programming, proceedings of eurogp'2000*, volume 1802 of *LNCS*, pages 44–59, Edinburgh, 15-16 April 2000. Springer-Verlag.

5. J. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.
6. J. Koza, F. Bennett III, J. Hutchings, S. Bade, M. Keane, and D. Andre. Rapidly reconfigurable field-programmable gate arrays for accelerating fitness evaluation in genetic programming. In J. Koza, editor, *Late breaking papers at the 1997 genetic programming conference*, pages 121–131, Stanford University, CA, USA, 13–16 July 1997. Stanford Bookstore.
7. W. Langdon and R. Poli. Why ants are hard. In J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. Goldberg, H. Iba, and R. Riolo, editors, *Genetic programming 1998: proceedings of the third annual conference*, pages 193–201, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.
8. D. Levi and S. Guccione. Genetic FPGA: Evolving Stable Circuits on Mainstream FPGA Devices. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 12–17. IEEE Computer Society, July 1999.
9. P. Martin. A Hardware Implementation of a Genetic Programming System using FPGAs and Handel-C. *Genetic Programming and Evolvable Machines*, 2(4):317–343, 2001.
10. J. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. Langdon, J. Miller, P. Nordin, and T. Fogarty, editors, *Genetic programming, proceedings of eurogp'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 April 2000. Springer-Verlag.
11. P. Nordin and W. Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic algorithms: proceedings of the sixth international conference (icga95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.
12. S. Perkins, R. Porter, and N. Harvey. Everything on the chip: a hardware-based self-contained spatially-structured genetic algorithm for signal processing. In J. Miller, A. Thompson, P. Thomson, and T. Fogarty, editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 165–174, Edinburgh, UK, 2000. Springer-Verlag.
13. D. Scott, S. Seth, and A. Samal. A Hardware Engine for Genetic Algorithms. Technical Report UNL-CSE-97-001, University of Nebraska-Lincon, Dept Computer Science and Engineering, University of Nebraska-Lincon., 4 July 1997.
14. B. Shackleford, G. Snider, R. Carter, E. Okushi, M. Yasuda, K. Seo, and H. Yasuura. A High Performance, Pipelined, FPGA-Based Genetic Algorithm Machine. *Genetic Programming and Evolvable Machines*, 2(1):33–60, Mar. 2001.
15. A. Thompson. Silicon evolution. In J. Koza, D. Goldberg, D. Fogel, and R. Riolo, editors, *Genetic programming 1996: proceedings of the first annual conference*, pages 444–452, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
16. G. Tufte and P. Haddow. Prototyping a GA pipeline for complete hardware evolution. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 18–25. IEEE Computer Society, July 1999.
17. Y. Yamaguchi, A. Miyashita, T. Marutama, and T. Hoshino. A Co-processor System with a Virtex FPGA for Evolutionary Computation. In R. Hartenstein and H. Grunbacher, editors, *10th International Conference on Field Programmable Logic and Applications (FPL2000)*, volume 1896 of *Lecture notes in Computer Science*, pages 240–249. Springer-Verlag, Aug. 2000.