Genetic Programming of Process Decomposition Strategies for Evolvable Hardware

Ho-Sik Seok, Kwang-Ju Lee, Byoung-Tak Zhang
Artificial Intelligence Lab (SCAI)
School of Computer Sci. and Eng.
Seoul National University
Seoul 151-742, Korea
{hsseok, kjlee, btzhang}@scai.snu.ac.kr

Dong-Wook Lee, Kwee-Bo Sim
Robotics and Intelligent Info. Syst. Lab
Electrical and Electronic Engineering
Chung-Ang University
Seoul 156-756, Korea
{dwlee, kbsim}@rics.cie.cau.ac.kr

Abstract

Evolvable hardware is able to offer considerably higher performance than general-purpose processors and significantly more flexibility than ASICs. In order to take the advantages of general-purpose processors and ASICs, dividing a complex process into subprocesses is essential. In this paper, we propose a evolutionary method called context switching that splits a task into a set of subtasks whose complexity is manageable on the given hardware. The method is based on genetic programming. Due to its expressive power, genetic program can represent flexible strategies for decomposing complex tasks. The effectiveness of context switching is demonstrated on the design of adaptive controllers for a team of autonomous mobile robots.

1. Introduction

General purpose processors provide a generic computational hardware for diverse applications. However, due to their architecture, they may show inferior computational performance. In contrast, application-specific integrated circuits (ASICs) exploit intrinsic characteristics of an application's algorithm that lead to a high performance. However, their algorithm mapping restricts the range of applicatons.

Evolvable hardware represents a hybrid approach of general purpose processors and ASICs. Evolvable hardware can offer considerably higher performance than general-purpose processors and more flexibility than ASICs [1]. In the hybrid approach, dividing a process is essential. This paper describes an evolutionary method for dividing a complex process into a sequence of subprocesses. The method is called *context switching* and based on genetic programming. Genetic programming is an automatic programming method that finds the best fit strategies (or programs) by means of natural selection and genetics [2]. Because the programs are usually represented as trees or S-expression of LISP, genetic programming can generate programs for solving the wide range of problems.

Context switching is inspired by the representational flexibility of genetic programming. We design this method to find a way to implement a process that is too complex to be represented on the hardware at hand. The evolved genetic tree represents the decomposition strategies of complex processes. Each subtree in the tree is a subprocess whose complexity (size) is manageable on the given hardware. These subtrees are implemented on the hardware by their computational sequence. To improve crossover and facilitate evolution of composite behaviors, we adopt results of other researchers.

The proposed method is tested on the design of controllers for a team of autonomous mobile robots. The world for this problem consists of obstacles, two KHEPERA robots and an object to transport. The object of genetic programming is to evolve a control logic so that the robots can transport the object to goal. By combining context switching and dynamic reconfigurability of evolvable hardware, we succeeded in evolving a simple adaptive learning system that can tune itself to more acceptible state.

The paper is organized as follows. Section 2 reviews related works on evolvable hardware. Section 3 illustrates our method in the context of genetic programming. Section 4 presents the context switching method in detail. Section 5 explains autonomous robot control problem. In Section 6, implementation details are described and some experimental results are shown. Section 7 summarizes the result and points out some future work.

2. Evolvable Hardware

Evolvable hardware is reconfigurable hardware whose configuration is under the control of an evolutionary al-

0-7695-0762-X/00 $10.00 © 2000 IEEE

gorithm [3]. Many researchers prefer genetic algorithms. Naito used genetic algorithms to evolve control logic circuit for an autonomous mobile robot, though they did not evolve circuit on evolvable hardware [4]. Iba evolved hardware logic by regarding the architecture bits of a PLD as a chromosome for genetic algorithms [5]. Miller used genetic algorithms to evolve a 3-bit binary multiplier based on the architecture of the XILINX 6216 FPGA [6]. So, it seems that the term Evolvable Hardware has been coined by the realization of genetic algorithm in reconfigurable hardware [7]. Why do researchers prefer genetic algorithms? The reasons are as following. First, genetic algorithms can use the configuration bit stream of a base architecture as chromosome. Second, genetic algorithms are simple and practically blind mechanisms of Nature. Third, genetic algorithms can be easily realized on hardware [8].

But, there are some researchers who attempted to use genetic programming as an evolutionary mechanism. Koza used genetic programming to evolve sorting network. He used evolvable hardware to reduce computation time [9]. Bennett III evolved a 60-decibel OP amp, he evolved circuit design by simulation [10]. Sakanashi applied genetic programming to digital circuit design. He used genetic programming to improve the hardware description in binary decision diagrams [11]. The advantage of genetic programming over genetic algorithms is that its function nodes can be used as function blocks of a circuit design. Therefore, a search based on genetic programming can remove unnecessary search step of genetic algorithms - evolution of function block from randomly distributed configuration bit stream. But it is very difficult to implement tree-structured chromosome on hardware. Therefore there is a need to find a method that can overcome this problem.

It is worthwhile commenting on hardware evolution mode. Translating a candidate genotype into a circuit for fitness evaluation can be achieved in software or hardware. Translating a genotype in simulation and in hardware is named as extrinsic and intrinsic modes of hardware evolution respectively [12]. Extrinsic evolution has the following disadvantages. First, extrinsic hardware evolution is slower than intrinsic hardware evolution by its computational expense in modeling and evaluating the evolved circuits. Second, circuits evolved extrinsically may not work in reality if they rely on unusual configuration of components. Third, by exploiting continuous-time dynamics and subtle physical effects, intrinsic hardware evolution can produce circuits beyond the scope of the human designer. But extrinsic evolution can not overcome the limit which was set by simulator designer [12]. So, many researchers use intrinsic method for their design [13, 14, 15].

Evolvable hardware is suitable for adaptive systems, fault-tolerant systems, and design automation by its dynamic reconfigurability. Adaptive systems are hardware systems that can reconfigure themselves through learning so as to adapt to environmental or job change. For the automated synthesis of CMOS circuits, Stoica introduced an approach based on the evolution on Programmable Transfer Arrays (PTA) [16]. By adaptation, he aimed to lengthen long-life survivability of remote interplanetary space mission system. His group demonstrated the possibility of adaptive systems which can reconfigure themselves on specified hardware. Kajitani made a controller of myoelectric artificial hand on evolvable hardware that has adaptation ability. His group showed that evolvble hardware based genetic algorithms outperformed neural-network based method and reduced learning time considerably [17]. Tanaka made a data compression chip for electrophotographic printing by using evolvable hardware. His group used evolvable hardware to compress(decompress) image data efficiently and quickly [18]. Fault-tolerant systems based on evolvable hardware is also interesting research area. Most researchers provide redundant blocks to assure redundancy. Ortega-Sanchez attempted to guarantee robustness by providing enough memory for each cell. In his work, each cell has enough memory to maintain a copy of the configuration register so it will be able to replace any other in case of failure [19]. Macias opened the possibility of fault-tolerant in his PIG (Processing Integrated Grid) architecture. In the PIG architecture, each cell can reconfigure itself to any of its neighboring cells [20]. Moreno proposed the FIPSOC (Field Programmable System On a Chip) which has self-repairing ability. The basic principle of the self-repairing strategy consists of using the array of digital macro cells in *dynamic reconfigurable mode* and duplicating the functionality of the system in both hardware contexts [21]. There are many researchers who use evolvable hardware for the design automation. Thompson used intrinsic hardware evolution to design an electronic circuit automatically under the control of genetic algorithms. He evolved a circuit to discriminate between square waves of 1kHz and 10kHz [14]. Miller evolved two-bit adder and two-bit multiplier automatically [22].

Recently, some researchers try to build a novel architecture composed of identical local cells that can be reconfigured to one of many predefined functions. Miller proposed reconfigurable arithmetic FPGA (RA-FPGA). He made a multi-functional component called Multiply-Divide-Logic (MDL). MDL can be reconfigured to perform one of the predefined functions. By using MDL structure, he could construct a FPGA architecture which executes intensive computation without sacrificing flexibility [23]. Macias proposed PIG architecture. His architecture is composed of massively-parallel, fine-grained, and self-reconfigurable cells. In his architecture, any local cell can control the mode of any neighboring cell and be reconfigured to any neighboring cell. By regular internal structure, he allowed

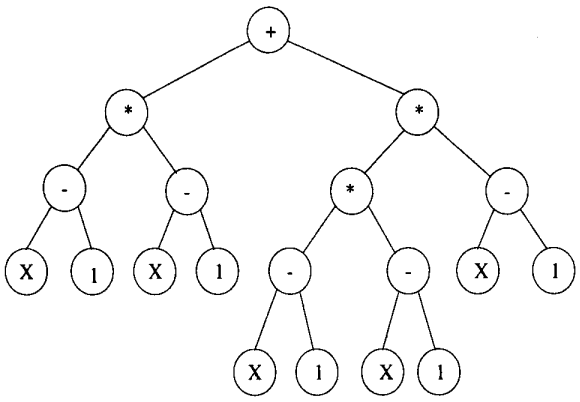

Figure 1. An example of genetic tree that represents $(x-1)^2+(x-1)^3$. Genetic programming uses tree structured choromosome which corresponds to certain process or a program.

systems to perform parallel computation with fault-tolerant ability [20].

3. Genetic Programming for Hardware Implementation

Genetic programming is a stochastic search method suitable for addressing inductive learning tasks. Genetic programming uses tree-structured chromosome inspired by the functional programming of LISP [24, 25]. An illustrative example of the genetic program is shown in Figure 1. Tree-structured form represents the interpretation process of input data. Genetic programming searches possible solutions as follows.

1. Population of chromosomes are randomly created to represent a pool of candidate solutions.
2. Fitness values are assigned to them based on how close they come to the desired solution.
3. Chromosomes are selected based on their fitness value and they are modified using genetic operators such as crossover and mutation.
4. Selected chromosomes comprise a new generation and new fitness values are assigned.

These steps are repeated until a termination condition is satisfied. Followings are more detailed explanation associated with our application - autonomous robot control problem. Preparatory steps for genetic programming run begin with the definition of function set and terminal set.

- Function Set
  Function set interprets sensory data.
  - *IF-OBJ* executes its left subtree if robot finds object, but otherwise executes its right subtree.
  - *IF-GOAL* executes its left subtree if robot finds goal identified as light source, but otherwise executes its right subtree.
  - *IF-FORWARD* executes its left subtree if robot finds goal and object is located between goal and front of the robot, but otherwise executes its right subtree. This node determines whether to push the object or to rotate around the object.
  - *IF-OBS 1~4* executes its left branch if one of four sets of sensor indicates the existence of obstacles, but otherwise executes its right subtree.
- Terminal Set
  Terminal set corresponds to robot actions.
  - *MOVE FORWARD* makes the robot move forward in the current direction (the direction of robot's linear vision sensor)
  - *TURN LEFT and MOVE FORWARD* changes the direction of the robot by 90 degree counter-clockwise and moves forward.
  - *TURN RIGHT and MOVE FORWARD* changes the direction of the robot by 90 degree clockwise and moves forward.
  - *MOVE BACKWARD* makes the robot move backward.
  - *TURN LEFT* changes the direction of the robot by 90 degree counter-clockwise.
  - *TURN RIGHT* changes the direction of the robot by 90 degree clockwise.
  - *RANDOM* executes one of 6 above functions at random.
- Crossover and Mutation

Crossover is a major operator of genetic programming. The traditional view is that crossover is primarily responsible for improvements in fitness [27]. Figure 2 shows an example of crossover. Crossover swaps two randomly chosen subtrees from two parent trees. Crossover is essential for generating a solution program in genetic programming. But, it has a serious drawback - the normal crossover operator selects a crossover point randomly regardless of its position within the tree structure. Thus crossover may destroy the building blocks [28]. To overcome this problem, Ito used depth-dependent crossover in which a crossover point is determined by a depth selection probability [28]. We

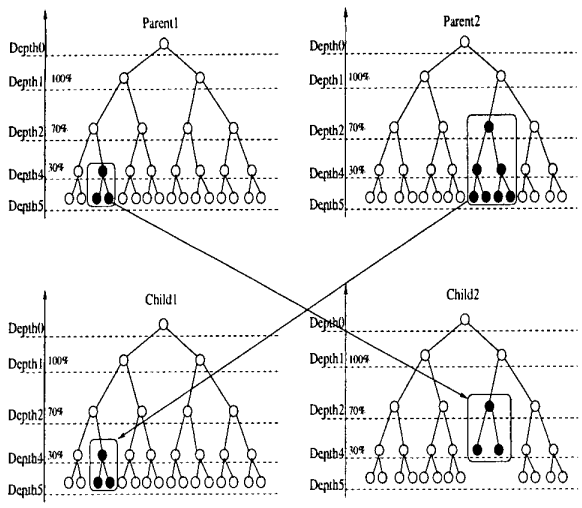

Figure 2. Simple depth-dependent crossover. Depth-dependent crossover assigns crossover probability to each depth. Crossover point is determined according to this probability. (The modification of figure of figure 16.1 in [28].)

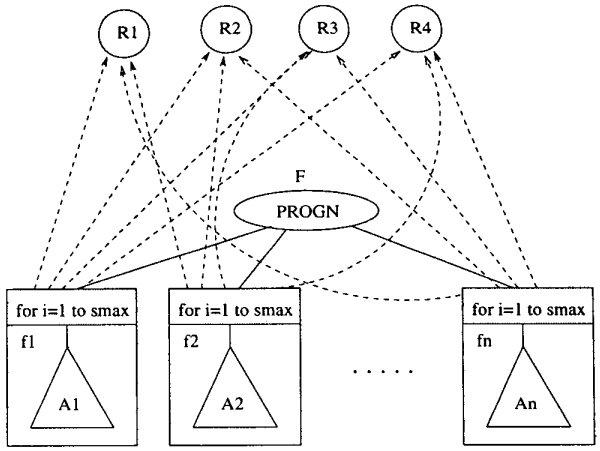

Figure 3. Schematic diagram for genetic programming with fitness switching. For multi process control, each subtree A_i represents a control program for a micro-behavior of each process R_i. The fitness of subtree A_i is assigned by executing it S_{max} times and measuring the goodness of its behavior by fitness function f_i. (Figure 18.2 in [30].)

modify depth-depedent crossover of Ito. In the method of Ito, crossover probability is tuned but in our simple depth-dependent crossover, crossover probability is assigned before execution and remains unchanged during experiment. New individual can exceed the depth limit. In this case, the extra part should be removed to keep the depth limit. To remove the extra part, we select one subtree of the extra part and swap the selected tree with the extra part. Sub-tree selection is performed randomly.

Mutation replaces a randomly selected node. Koza has argued that mutation is in fact useless in genetic programming because of the position-independence of genetic programming subtrees, and the large number of chromosome positions in typical genetic programming populations [2]. But mutation is a useful operator for finding hidden nodes and often works better in small populations, depending on the domain [27].

- Fitness Function

Fitness measures how well a program has learned to predict the outputs from the inputs [29]. Programs such as robot control require at least two kind of fitness function. But, existing genetic programming methods are not practical enough to find an optimal solution in this domain. To evolve this kind of complex behavior we adopt a method called *fitness switching* [30]. Fitness switching is a method for evolving complex behaviors with genetic programming. It is based on the incremental learning procedure. The procedure for applying fitness switching to a specific problem can be summarized as follows.

1. Define the primitive actions for the problem domain. These are the terminal set, i.e. the actions executed by the agents to solve the problem.
2. Define a small number of micro-behaviors
   B=$\{B_1, B_2, ..., B_n\}$
   that constitute the original problem-solving behavior.
3. Define a fitness function for each micro-behavior.
   F=$\{f_1, f_2, ..., f_n\}$
4. Design a sequence of micro-behaviors S_t or their combinations to achieve the target behavior:
   $S_t = S_{t-1} \oplus B_t$
   where $\oplus$ denotes the append operator and S_0 is the empty sequence, i.e. S_0 =<>. The corresponding sequence of fitness functions are defined by
   $F_t = F_{t-1} \oplus f_t$
5. Define the structure of a genetic program A as having n subtrees, $A^i(i = 1, ..., n)$, immediately under the root node. A=$\{A^1, A^2, ..., A^n\}$

After all micro-behaviors are evolved, evolved trees constitute a large tree. This large tree is the desired solution

[30]. For our autonomous robot control problem, we need to evolve two kinds of micro-behavior: *herding* and *homing*. *Herding* means finding an object that will be transformed to goal by robots. For *herding*, *equation 1* is used as fitness function. *Homing* means transforming object to goal. For *homing*, *equation 2* is used as fitness function.

$f_1 : F_{new} = F_{old} + w_1 \times (\#collisions) + w_2 \times (\#steps)$ - *equation 1*

$f_2 : F_{new} = F_{old} + w_1 \times (\#miss) + w_2 \times (\#steps) + vision \times w_3$ - *equation2*

Collisions means the number of collision with obstable, *miss* means the number of miss from the goal, *steps* means the number of step, and *vision* means the value of the linear vision sensor.

- Selection Method

After the fitness of each chromosome has been assigned, it should be determined whether to keep it in the population or to replace it. For this task, the selection operator is used. There are various selection operators. Fitness-proportional selection specifies probabilities. These values give a chance to pass offspring into the next generation. A chromosome i is given a probability of *equation 3* for being able to pass on traits [29].

$P_i = f_i / \sum_i f_i$ - *equation 3*

Ranking selection is based on the fitness order. The selection probability is assigned to individuals as a function of their rank in the population. Tournament selection is based on competition in a *subset* of the population. The number of individuals, called the tournament size, is selected randomly, and a selective competition takes place. The traits of the better individuals in the tournament are then allowed to replace those of the worse individuals [29]. For our problem, fitness-proportional selection operator is used.

4. Context Switching

Context switching is a method designed for dividing a process automatically for evolvable hardware application. This procedure is designed for adaptive system and use tree structured chromosome of genetic programming. Context switching is composed of the following components.

- *Evolver* evolves a decomposition strategies using function nodes and terminal nodes
- *Hardware library* maintains the information for hardware implementation such as gate, routing etc.

The procedure for applying context switching to a specific problem can be summarized as follows.

1. Specify the hardware resource at hand.

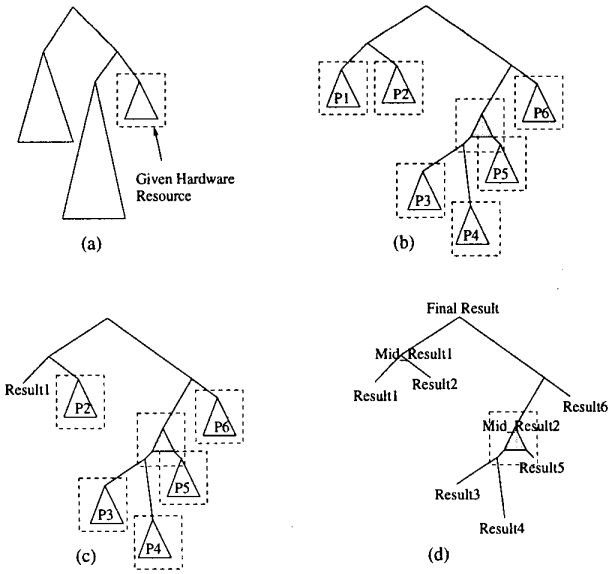

Figure 4. Context switching procedure. Dashed square box represents a given hardware resource. By context switching, the tree is divided into subtrees whose size is representable on hardware.

2. Define the function set and the terminal set for the task.
3. Define the condition nodes. These nodes are elements of function set. But genetic operators such as crossover and mutation are not applied to these nodes. These nodes are used as *flag*. The logic on hardware is not replaced until the condition node is set.
4. Evolve a genetic tree using the function set and the terminal set using evolver. The evolved tree represents the decomposition strategies of the task.
5. Divide the evolved tree into subtrees whose size are representable on the given hardware. The division step does not dive into automic process. If the remaining subtrees are representable on the given hardware, tree division procedure stops at that depth (Stage (b) of Figure 4).
6. Convert the divided subtrees into hardware library. This library has the information for implementing hardware logic (gate and routing information).
7. A subtree which will be located on hardware is selected by their sequential computation order. The functionality of the selected tree is represented on hardware and outputs computation results according to given data (Stage (c) of Figure 4).

8. After all subtrees are computed, the midresults are combined to produce the final result (Stage (d) of Figure 4).

This method is called context switching since a genetic tree is divided into a set of subtrees whose size is representable on the given hardware and the context on the hardware is switched by computational order. In order to use context switching, two factors - *relation complexity* and *dwelling period* - should be considerd.

- *Relation complexity* means the complexity of relation between subprocesses. i.e. the existence of cycle, the depth of recursion etc.
- *Dwelling period* means the minimum duration of a subprocess on the hardware among subprocesses.

If a process has long dwelling period and low relation complexity, context switching can be applied to this process easily. If a process has more shorter dwelling period and higher relation complexity, context swtiching to this process becomes more difficult. Because it has long dwelling period and low relation complexity, we choose the robot problem to verify our method more easily.

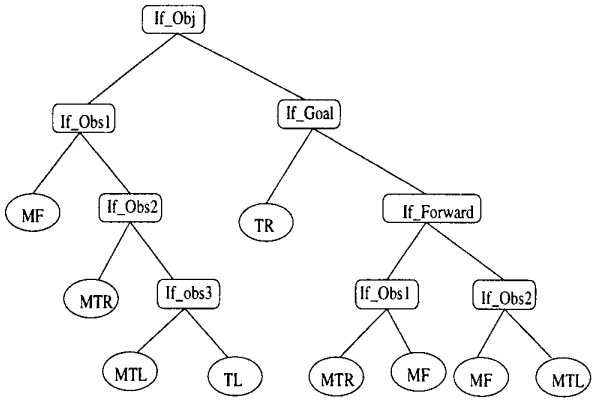

Figure 5. An example of control structure. At root, robot determines whether it reaches the object pushed to the goal. If it finds the object, control authority goes to the right subtree, else to the left subtree. At one of two subtrees, robot determines whether it confronts an obstacle. After this procedure, robot comes to one of terminal nodes whose information is used as robot action.

5. Autonomous Robot Control Problem

Robotics has developed systems that are able to automate simple and repetitive tasks. These robots are mostly programmed in a very explicit way so that the environment of the robot and all the operations are described in previous stage. But in the case of the mobile applications, the environment is perceived via sensor system of robot. The perceived data is far different from well-defined environmental data of usual robots. Thus, in order to preserve its autonomy, the mobile robot must retain some degree of deciding ability in a complex and dynamic environment. One of the objects in AI and Aritificial Life research is building an autonomous system that can adapt to the changing and often unpredictable world. Building autonomous robot is a good example of such an autonomous system research [31].

There are many attempts to control autonomous robot using various algorithms. Nolfi used a modular neural network architecture that clearly outperformed other architecture in performing a task of garbage collection [32]. Kalmar tried to combine module-based reinforcement learning with robot-learning problem [33]. Naito evolved a logic circuit that controls an autonomous mobile robot using genetic algorithms [4]. Koza used computer simulation to evolve a control program for a grid based mobile robot to mop an area containing obstacles using genetic programming [34]. Wilson evolved hierarchical behaviors to locate a goal object in a maze for a mobile robot [35]. Lee made a box-pushing robot using genetic programming. The robot pushed an object to a goal in non-obstacle environment [36]. Our problem is similar to Lee's problem. But our work is more difficult than Lee's work, because our job is finding a way to push an object by two robots avoiding obstacles.

Figure 5 shows a control structure of an autonomous mobile robot using genetic tree. Terminal nodes represent the movement of a robot and function nodes represent the interpretation of sensor data. A path from root node to one of leaf nodes is an interpretation path of a sensor pattern. A genetic tree must have sufficient depth to provide all the possible interpretation paths. But without the depth limit, a tree can grow to an unacceptably big size by addition of redundant nodes - known as the *intron* problem [37]. In the case of hardware implemented genetic programming, the intron problem may have a severe drawback. With *introns*, the precious hardware resources are used for unnecessary computataion. Therefore we limit the tree depth.

6. Implementation and Experiments

To make context switching easy, there are two kinds of blocks: *block1* for function node and *block2* for terminal node. Figure 6 shows the block for function node. At initial state, block1 have the structure shown in Figure 6. When the selected subprocess is located, the * printed gates are changed to represent one of the predefined functions. By

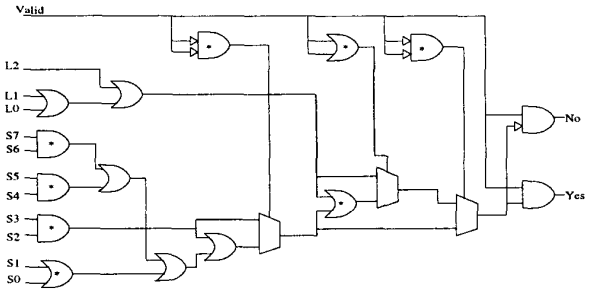

Figure 6. This block is for function node. By changing (*) printed gate, this block is converted to one of predefined functions.

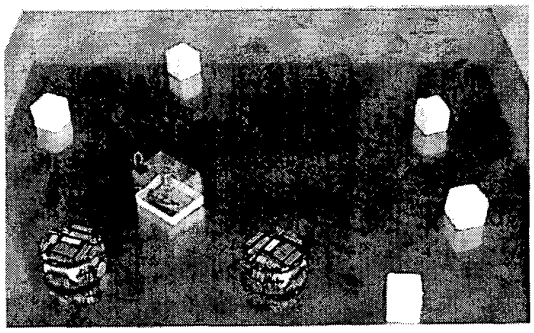

Figure 7. The experimental environment. There are two KHEPERA robots, one object and some obstacles. Our problem is to find an evolutionary strategy that makes the robots cooperate to push the object while avoiding collision with obstacles.

converting and routing these blocks, the selected subprocess is converted to tree structured hardware logic.

Figure 8 shows abstract system diagram. The system consists of the host PC part and the KHEPERA robot part. At the KHEPERA part, robot sends environmental data to host PC and receives motor control data from host PC. The host PC part is composed of robot control unit, communication unit and evolvable hardware unit. Robot control unit interprets sensor data and computes fitness value based on sensor data. Communication unit relayes sensor data and behavior data between robot control unit and evolvable hardware unit. Also communication unit stores population and generates fitness graph. Evolvable hardware unit controls evolvable hardware. At this unit, genetic operations are performed to generate the next generation. Selection of behavior according to sensor data is performed at this unit.

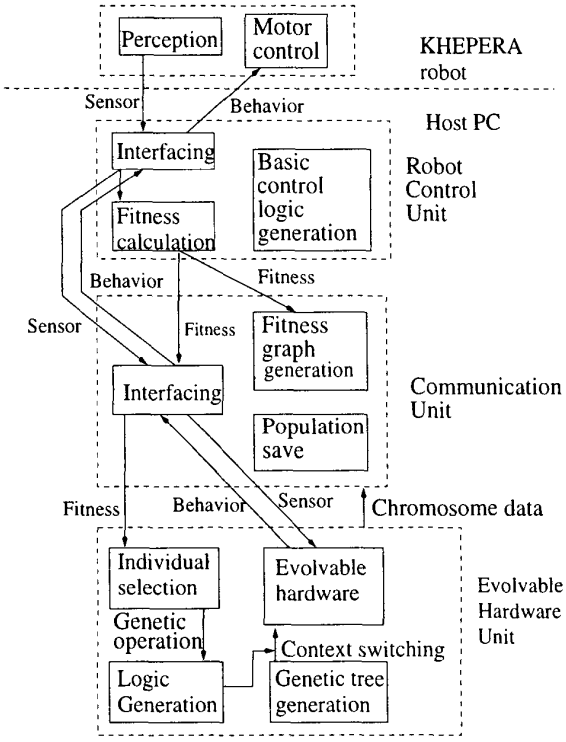

Figure 8. System diagram

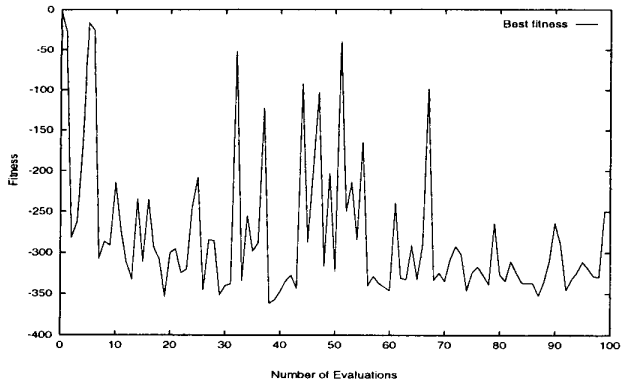

Figure 9. The best fitness

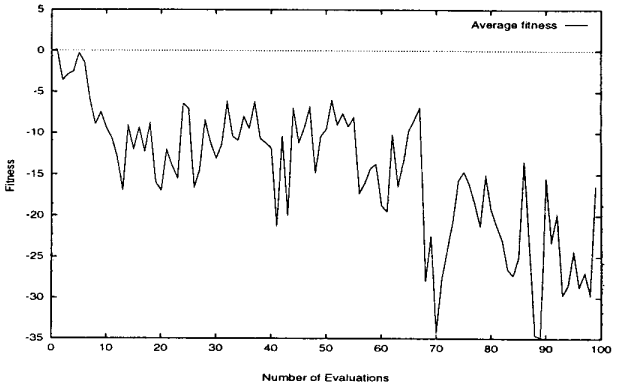

Figure 10. The average fitness

For the experiment, we use XC6216 FPGA of XILINX. It is mounted on H.O.T. board of V.C.C. co. Figure 7 shows experimental environment. There are two KHEPERA robots, one object, and some obstacles. Our object is finding an evolutionary algorithms that makes the robots cooperate in pushing the object to goal avoiding collisions with obstacles. Figure 9 and 10 show the best fitness and the average fitness during 100 generations. Figure 11 shows the changes in the number of collisions and successes per generation. The best fitness graph shows comparably low value at generation 10. But the best fitness is unstable and is stablized after generation 70. The reason for the large deviation is that the fitness value is also affected by randomness. But after generation 70, the number of successes increase continuously. Our implementation has a problem. It can not generate an environment map. Without map, the learning of a robot is restricted to the discovery of outputs to each input patterns. Therefore, the robot can not find the optimal path to the goal and commits same error repeatedly. Of cause, the robot succeeded in finding the behavior strategy to accomplish its task. However, the function for map generation is essential to improve performance.

By using context switching, we got two results.

1. We found a way to implement a process whose size is bigger than the given hardware

2. We have shown the posiibility of hardware that can reconfigure itself by adaptiveness. Due to the learning ability of genetic programming, the tree keeps tuning itself during run.

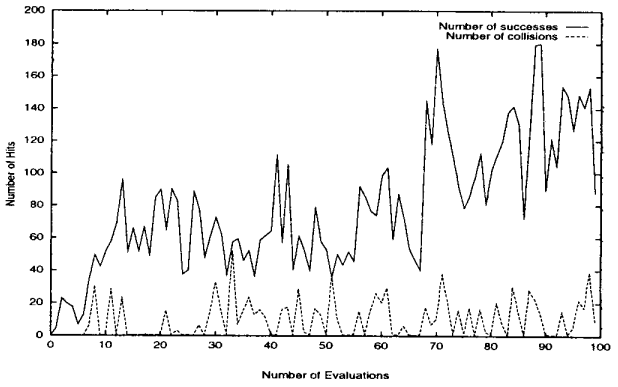

Figure 11. The number of collisions and successes

7. Conclusion

We presented a genetic programming method for automatically decomposing complex tasks into subtasks for easy implementation on evolvable hardware. The method

is motivated by the observation that genetic trees can be intrepreted as a set of processes. In the context of the object transportation problem, we have shown that context switching with fitness switching can solve problems with a reasonable size of solutions. But we came to the conclusion that the robots should have a map generator to improve the performance.

Our main findings from this work can be summarized as follows. First, we found a way to represent a process that is too big to be implemented on the hardware at hand. Second, we have shown the possibility of hardware that learns adatively and configures itself through learning.

The ultimate aim of our research is to find a way to implement a system that is capable of adaptive learning. For that purpose, evolvable hardware provides a useful feature of dynamic reconfigurability. Through a simple robot experiment, we demonstrate that the combination of context switching and evolvable hardware can be useful for adaptive system. To prove the usefulness of context switching, further research in a more realistic setting is required.

Acknowledgements

This research was supported by the Korea Science and Engineering Foundation (KOSEF) under grant 981-0920-107-2.

References

[1] G. Lu, H. Singh, M.H. Lee, N. Bagherzadeh, F. J. Kurdahi, E. M. C. Filho, and V. C. Alves, The MorphoSys Dynamically Reconfigurable System-On-Chip, *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pp. 152-160, 1999.

[2] J. R. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*, Cambridge, MA, USA, MIT Press, 1992.

[3] A. Stoica, D. Keymeuler, R. Tawel, C. Salazar-Lazaro, and W. Li, Evolutionary Experiments with a Fine-Grained Reconfigurable Architecture for Analog and Digital CMOS Circuits, *Proceedings of the first NASA/DoD workshop on Evolvable Hardware*, pp. 76-84, 1999.

[4] T. Naito, R. Odagiri, Y. Matsunaga, M. Tanifuji, and K. Murase, Genetic Evolution of a Logic Circuit Which Controls an Autonomous Mobile Robot, *Proceedings of the First International Conference on Evolvable Systems*, pp. 210-219, 1996.

[5] H. Iba, M. Iwata, and T. Higuchi, Machine Learning Approach to Gate-Level Evolvable Hardware, *Proceedings of the First International Conference on Evolvable Systems*, pp. 327-343, 1996.

[6] J. F. Miller and P. Thomson, Aspects of Digital Evolution: Geometry and Learning, *Proceedings of the Second International Conference on Evolvable Systems*, pp.25-35, 1998.

[7] H. DeGaris, Evolvable Hardware: Principles and Practice, *Communication of ACM*, August 1997.

[8] M. Perkowski, A. Chebotarev, and A. Mishchenko, Evolvable Hardware or Learning Hardware? Induction of State Machines from Temporal Logic Constraints, *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pp.129-138, 1999.

[9] J. R. Koza, F. H. Bennett III, J. L. Hutchings, S. L. Bade, M. A. Keane, and D. Andre, Evolving Computer Programs using Rapidly Reconfigurable Field-Programmable Gate Arrays and Genetic Programming, *Proceedings of the ACM Sixth International Symposium on Field Programmable Gate Arrays*, pp. 209-219, 1998.

[10] F. H. Bennett III, J. R. Koza, D. Andre, and M. A. Keane, Evolution of a 60 Decibel Op Amp Using Genetic Programming, *Proceedings of the First International Conference on Evolvable Systems*, pp. 455-469, 1996.

[11] H. Sakanashi, T. Higuchi, H. Iba, and Y. Kakazu, Evolution of Binary Diagrams for Digital Circuit Design Using Genetic Programming, *Proceedings of the First International Conference on Evolvable Systems*, pp. 470-481, 1996.

[12] P. Layzell, Reducing Hardware Evolution's Dependency on FPGAs, *Proceedings of the Seventh International Conference on Microelectronics for Neural, Fuzzy and Bio-inspired Systems*, pp. 171-178, 1999.

[13] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K.Toda, M. Salami, N. Kajihara, and N. Otsu, Real-world Applications of Analog and Digital Evolvable Hardware, *IEEE transactions on evolutionary computation*, vol3, no 3, pp. 220-235, se ptember 1999.

[14] A. Thompsom, An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics, *Proceedings of the First International Conference on Evolvable Systems*, pp. 390- 405, 1996.

[15] P. Layzell. A New Research Tool for Intrinsic Hardware Evolution, *Proceedings of the Second International Conference on Evolvable Systems*, pp. 47-65, 1998.

[16] A. Stoica, D. Keymeulen, C. Lazaro, W. Li, K. Hayworth, and R. Tawel, Toward On-board Synthesis and Adaptation of Electronic Functions:An Evolvable Hardware Approach, *Proceedings of 1999 Aerospace Conference*, pp. 351-357, 1999.

[17] I. Kajitani, T. Hoshino, D. Nishikawa, H. Yokoi, S. Nakaya, T. Yamauchi, T. Inuo , N. Kajihara, M. Iwata, D. Keymeulen, and T. Higuchi, A Gate-Level EHW chip: Implemeting GA Operations and Reconfigurable Hardware on a single LSI, *Proceedings of the Second International Conference on Evolvable Systems*, pp. 1-12, 1998.

[18] T. Hikage, H. Hemmi, and K. Shimohara, Compression of Evolutionary Methods for Smoother Evolution, *Proceedings of the Second International Conference on Evolvable Systems*, pp. 115-124, 1998.

[19] C. Ortega-Sanchez and A. Tyrell, Fault-Tolerant Systems: The Way Biology Does It, *Proceedings of the 23rd Euromicro conference*, pp. 146-151, 1997.

[20] N. J. Macias, The PIG Paradigm:The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture, *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pp. 175-180, 1999.

[21] J. M. Moreno, J. Madrenas, R. Kielbik, J. Faura, and J. M. Insenser, Realization of Self-Repairing and Evolvable Hardware Structures by Means of Implicit Self-Configuration, *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pp. 182-187, 1999.

[22] J. F. Miller and P. Thompson, Discovering Novel Digital Circuits using Evolutionary Techniques, *Proceedings of IEE Half-day Colloquium on Evolvable Hardware Systems*, 1998.

[23] N. L. Miller, and S. F. Quigley, A Reconfigurable Integrated Circuit for High Performance Computer Arithmetic, *Proceedings of IEEE International Symposium on Circuits Systems*, 1999.

[24] N. I. Nikolaev, H. Iba, and V. Slavov, Inductive Genetic Programming with Immune Network Dynamics, *Advances in Genetic Programming 3*, MIT Press, pp. 355-376, 1999.

[25] D. Andre and A. Teller, A Study in Program Response and the Negative Effects of Introns in Genetic Programming, *Proceedings of Genetic Programming 1996*, pp.12-20, 1996.

[26] J. R. Koza and F. H. Bennett III, Automatic Synthesis, Placement, and Routing of Electrical Circuits, *Advances in Genetic Programming 3*, MIT Press, pp. 105-134, 1999.

[27] S. Luke and L. Spector, A Comparison of Crossover and Mutation in Genetic Programming, *Proceedings of Genetic Programming 1997*, pp. 240-248, 1997.

[28] T. Ito, H. Iba, and S. Sato, A Self-Tuning Mechanism for Depth-Dependent Crossover, *Advances in Genetic Programming 3*, MIT Press, pp. 377-399, 1999.

[29] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming an Introduction*, Morgan Kaufmann, 1998.

[30] B.-T. Zhang and D.-Y. Cho, Fitness Switching: Evolving Complex Group Behaviors Usign Genetic Programming, *Proceedings of Genetic Programming 1998*, pp. 431-438, 1998.

[31] M. Pattie, Behavior-based Artificial Intelligence, *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pp. 2-10, 1993.

[32] S. Nolfi, Using Emergent Modularity to Develop Control Systems for Mobile Robots, *Adaptive Behavior* vol.5, pp. 343-363, 1997.

[33] Z. Kalmar, C. Szepesvari, and A. Lorincz, Module-based Reinforcement Learning: Experiments with a Real Robot, *Machine Learning* vol.31, pp. 55-85, 1998.

[34] J. R. Koza, Obstacle-Avoiding Robot, *Genetic Programming II: Automatic discovery of reusable programs*, MIT Press, pp. 365-376, 1994.

[35] M. S. Wilson et al, Evolving hierarchical Robot Behaviors, *Robotics and Autonomous Systems*, pp. 215-230, 1997.

[36] W. P. Lee, J. Hallam, and H. H. Lund, Applying Genetic Programming to Evolve Behavior Primitives and Arbitrators for Mobile Robots, *Proceedings of IEEE International Conference on Evolutionary Computation*, pp. 501-506, 1997.

[37] D. Andre and A. Teller, A Study in Program Response and the Negative Effects of Introns in Genetic Programming, *Proceedings of Genetic Programming 1996*, pp. 12-20, 1996.