Asynchronous Evolution by Reference-Based Evaluation: Tertiary Parent Selection and Its Archive

Tomohiro Harada[1,2] and Keiki Takadama[1]

[1] The University of Electro-Communications,
1-5-1 Chofugaoka, Chofu, Tokyo, Japan
[2] Research Fellow of the Japan Society for the Promotion of Science DC, Japan
harada@cas.hc.uec.ac.jp, keiki@inf.uec.ac.jp
http://cas.hc.uec.ac.jp

Abstract. This paper proposes a novel *asynchronous reference-based evaluation* (named as ARE) for an asynchronous EA that evolves individuals independently unlike general EAs that evolve all individuals at the same time. ARE is designed for an asynchronous evolution by tertiary parent selection and its archive. In particular, ARE asynchronously evolves individuals through a comparison with only three of individuals (i.e., two parents and one *reference* individual as the tertiary parent). In addition, ARE builds an archive of good *reference* individuals. This differ from synchronous evolution in EAs in which selection involves comparison with all population members. In this paper, we investigate the effectiveness of ARE, by applying it to some standard problems used in Linear GP that aim being to minimize the execution step of machine-code programs. We compare GP using ARE (ARE-GP) with steady state (synchronous) GP (SSGP) and our previous asynchronous GP (Tierra-based Asynchronous GP: TAGP). The experimental results have revealed that ARE-GP not only asynchronously evolves the machine-code programs, but also outperforms SSGP and TAGP in all test problems.

Keywords: Genetic programming, asynchronous evolution, machine-code program.

1 Introduction

In general Evolutionary Algorithms (EAs) typified as Genetic Algorithm (GA) [1] and Genetic Programming (GP) [2] evolve individuals (solutions) by repeating a *generation* step. This approach waits for evaluations of all individuals and generates a next population through the parent selection and the individual deletion. This requires that all individuals are evaluated at the same time, i.e., it requires to wait for the slowest evaluation of a certain individual when the evaluation time of individuals differ from each other, which consumes a heavy computational time. To tackle this problem, *asynchronous* approaches have recently been proposed [3][4], that evolves individuals independently, i.e., individuals do not have to wait for the

M. Nicolau et al. (Eds.): EuroGP 2014, LNCS 8599, pp. 198–209, 2014.

evaluations of other individuals. As GP employing the asynchronous approach, we have proposed TAGP (Tierra-based Asynchronous Genetic Programming) [5] as one kind of machine-code GP based on the idea of a biological simulator, Tierra [6]. The advantages employing the asynchronous approach for machine-code GP are summarized as follows: (1) it can continue to evolve individuals (programs) even if individuals cannot complete their evaluation (e.g., due to an infinite loop), because it is not required to wait for the evaluation of all individuals, and (2) it increases the chance of selecting quickly evaluated individuals by evolving them immediately after completing their evaluations.

Our previous research [5] reported that TAGP can asynchronously evolve the machine-code programs even if they include loop structure. However, TAGP has the following two essential problems: (1) Unlike general EAs, TAGP cannot guarantee to select good individuals as the parents from a population, which prevents performance improvements, (2) since TAGP selects individuals as the parents or deletes them depending on a threshold based on an absolute evaluation, it is difficult to properly evolve individuals in the case that a proper threshold cannot be determined.

To overcome these problems, this paper proposes a novel asynchronous reference-based evaluation (named as ARE) for an asynchronous EA by *tertiary* parent selection and its *archive*, which not only inherits the advantage of TAGP but also overcomes its weak points.

In particular, in ARE, an *archive* mechanism employed which preserves good individuals to improve the performance. Parent are selected and individuals are deleted asynchronously through a comparison of the two parents with the *tertiary* parent that is randomly selected from an archive of good individuals. This step checks whether two parents are bad individuals. This is called a *reference-based evaluation*, a relative evaluation, which does not require a threshold as an absolute evaluation like TAGP. To investigate the effectiveness of ARE, this paper applies it to the Linear GP problems and compares GP using ARE (ARE-GP) with steady-state GP (SSGP) [7] as the synchronous GP and with TAGP as the asynchronous GP.

This paper is organized as follows. Section 2 introduces TAGP that we proposed and explains its problems. Section 3 proposes the novel asynchronous reference-based evaluation for an asynchronous EAs. Section 4 conducts the experiments for comparing the results of ARE-GP with those of SSGP and TAGP, and discuss their results. Finally, our conclusion is given in Section 5.

2 Tierra-Based Asynchronous Genetic Programming

2.1 Overview

TAGP (Tierra-based Asynchronous Genetic Programming) [5] is a kind of machine-code GP employing the asynchronous approach. TAGP is based on the idea of a biological simulator, Tierra [6]. It uses a system like Tierra to evolve programs that solves given tasks. Unlike Tierra, TAGP introduces fitness and executes parent selection and mechanisms that delete individuals based on fitness.

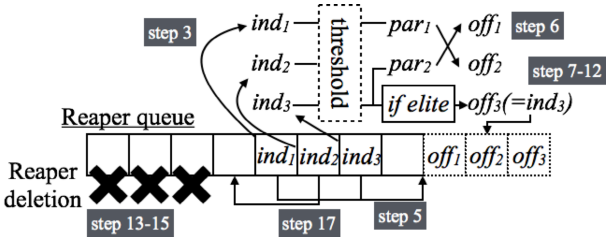


Fig. 1. An illustration of TAGP

2.2 Algorithm

Fig 1 shows an illustration of TAGP, while Algorithm 1 explains it further. In Algorithm 1, *ind* indicates an individual just after evaluation, $ind.f$ and $ind.f_{acc}$ respectively indicate the fitness and a fitness accumulated through a repeated evaluation process of *ind*, and MAX_POP indicates the maximum population size. All individuals are stored in a queue named as *reaper queue*. Firstly, each individual is evaluated in (pseudo-)parallel (step 1), and it accumulates its fitness to $ind.f_{acc}$, when its evaluation completes (step 2). If its accumulated fitness exceeds a certain threshold (e.g., the maximum fitness in Algorithm 1), the threshold is subtracted from the accumulated fitness and it generates offspring(step 3-6). For example, if the accumulated fitness of ind_1 and ind_3 exceed the threshold (the maximum fitness), they generate offspring in the genetic operators as shown in Fig. 1. And if *ind* has the maximum fitness and it is better than the individual that previously has the maximum fitness, *ind* is replicated as an *elite* individual without any genetic operations in order to preserve the good individuals in the population (step 7-12). For example, if ind_3 has the maximum fitness, it is replicated as the elite individual as shown in Fig. 1. The position of each individual in the queue changes depending on whether an individual can generate offspring or not. If an individual generate offspring, its position in the queue shifts toward lower (step 5), otherwise its position shifts toward upper (step 17). For example, in Fig. 1, since ind_1 and ind_3 generate their offspring, their positions in the queue shift toward lower, while since ind_2 cannot generate offspring, its position shifts toward upper. If the population size exceeds the maximum population size, the reaper queue mechanism removes the individual located at the top in the reaper queue (step 13-14). For example, when three offspring are added as shown in Fig 1, three individuals at the top of the reaper queue is removed.

The main feature of TAGP is summarized as follows: (1) Each individual in TAGP can be asynchronously evolved without waiting for other evaluation because the parents are selected to evolve their offspring only depending on their accumulated fitness, i.e., such parent selection is executed when the accumulated fitness of an individual exceeds a certain threshold, and (2) the reaper queue mechanism in TAGP can remove an individual that requires huge evaluation

Algorithm 1. An algorithm of TAGP

```
1. Evaluating fitness of ind (ind.f)
2. ind.f_acc ← ind.f_acc + ind.f
3. if ind.f_acc ≥ f_max then
4.    ind.f_acc ← ind.f_acc − f_max
5.    Shifting the position of ind toward lower
6.    Generating offspring of ind
7.    if ind.f = f_max then
8.       if ind.f is better than ind_elite.f then
9.          Replicate ind without genetic operations
10.      end if
11.      ind_elite ← ind
12.   end if
13.   if Population size>MAX_POP then
14.      Removing the individual at the top of the queue
15.   end if
16. else
17.   Shifting the position of ind toward upper
18. end if
```

time or does not complete its evaluation (e.g., because of an infinite loop) *before* completing its evaluation.

3 Asynchronous Reference-Based Evaluation

This paper proposes a novel *asynchronous reference-based evaluation* (named as ARE) for an asynchronous EA, which not only inherits the advantages of TAGP but also overcomes the problems of TAGP. In this section, we firstly explain the main concept of ARE, and then its algorithm.

3.1 Concept

The main concept of ARE is based on (1) the preservation of good individuals (i.e., *archive*) and (2) the deletion of bad individuals with quick evaluation with the *relative* evaluation which does not require a threshold as an *absolute* evaluation like in TAGP. Regarding the first issue (i.e., the good individuals preservation), it is generally difficult to guarantee to preserve the good individual in a population due to an asynchronous manner, which means that good individuals are not always to be selected as parents for an evolution. To keep good individuals, the *archive* is employed to preserve good individuals in ARE while deleting the individuals that (a) have low fitness or (b) require a huge evaluation time (or do not complete their evaluation). From the viewpoint of the low fitness, if the individuals are worse than the reference individual (described in the next paragraph), they become candidates for the deletion. From the viewpoint of the huge evaluation time, on the other hand, the reaper queue

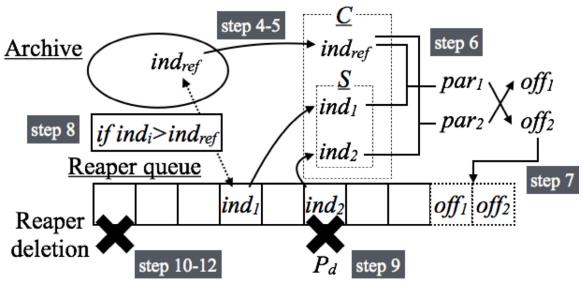


Fig. 2. An illustration of ARE

is employed to determine the individual that should be deleted using the same mechanism as TAGP.

Regarding the second issue (i.e., the deletion of bad individuals with quick evaluation with the *relative* evaluation), what should be noted here is that, in the asynchronous evaluation, the individuals that have high fitness are regarded as good like the general EAs, while individuals that complete their evaluation are also regarded as good. However, the individuals that quickly complete their evaluation do not always have good fitness. Due to such a feature, the individuals with quick evaluation are preferentially selected as parents regardless of its fitness. This results in the population being filled with offspring generated by such parent, and it is easy to fall into local optima through this kind of an evolution. To avoid such a situation, all individuals are compared with a high fitness individual that has already completed its evaluation. This is called the *tertiary* or *reference* individual. If they are worse than the reference individual, they are not selected as parents. This contributes to selecting the good individuals by excluding bad individuals whose evaluation time is short. In ARE, an evaluation based on a comparison with a reference individual is called as *relative* evaluation, which does not require a threshold as an *absolute* evaluation. Concretely, the parent selection and the individual deletion are asynchronously executed to evolve individuals through a comparison of the offspring with the *reference* individual.

Finally, the main difference between ARE and TAGP is summarized as follows: (1) TAGP requires the accumulated fitness and the selection threshold f_{max}, while ARE does not require these conditions, i.e., any fitness function in ARE can be employed like general EAs. This means that ARE can be applied to the problems where an optimal solution is unknown, and (2) TAGP does not have the archive mechanism, while ARE has it to guarantee to maintain the good individuals in the archive. This mechanism increases the selection pressure for better individuals by excluding bad individuals.

3.2 Algorithm

Fig. 2 shows an illustration of ARE, while Algorithm 2 explains it further. In Algorithm 2, *ind* indicates an individual just after evaluation. All individuals

Algorithm 2. An algorithm of ARE

1. Evaluating fitness of ind ($ind.f$)
2. $S \leftarrow ind$
3. **if** $|S| = 2$ **then**
4. Randomly selecting ind_{ref} from archive
5. $C \leftarrow S \cup \{ind_{ref}\}$
6. Selecting two individuals from C and generating offspring
7. Adding offspring into the bottom of the reaper queue
8. Replacing ind_{ref} with an individual in S that is better than ind_{ref}
9. Removing individuals in S that is worse than ind_{ref} in the probability P_d
10. **if** Population size>MAX_POP **then**
11. Removing individuals depending on the reaper queue
12. **end if**
13. $S \leftarrow \phi$
14. **end if**

are stored in either the reaper queue or the archive. Like TAGP, in ARE, the individuals are evaluated in (pseudo-)parallel (step 1), and the parent individuals are selected when two individuals complete their evaluations (here we call them *temporally-selected individuals*). One of the unique aspects of ARE, is that parent selection is done by the tournament selection from the temporally-selected individuals (S in Algorithm 2) and the reference individual (ind_{ref} in Algorithm 2) that is randomly selected from the archive (step 4-6) (Note that two individuals (not three) are selected in TAGP). This mechanism guarantees that individuals with high fitness will be mated with. After the parent selection, the two offspring generated from the two selected parents are added into the bottom of the reaper queue (step 7).

Other unique aspects of ARE are the use of an archive mechanism that preserves good individuals and the deletion mechanism limit the archive size. In order to determine the individuals that should be archived or should be deleted, the temporally-selected individuals are compared with the reference individual. If one of the temporally-selected individuals is better than the reference individual, it is archived and the reference individual change its position to the bottom of the reaper queue alternatively (step 8). To maintain the diversity of the individuals in the archive, if the better temporally-selected individual already exists in the archive and the reference individual is unique in the archive, they are not replaced each other. For example, if ind_1 is better than ind_{ref}, ind_1 is archived and ind_{ref} is added to the bottom of the reaper queue as shown in Fig. 2. This mechanism guarantees to preserve the good individuals in the archive. On the other hand, if the temporally-selected individuals are worse than the reference individual, they are removed from the reaper queue with a certain probability P_d (step 9) (This deletion is called *fitness deletion*, and the probability P_d is called *fitness deletion probability*). For example, if ind_2 is worse than ind_{ref}, ind_2 is removed from the reaper queue with the probability P_d with the fitness deletion as shown in Fig 2.

If the temporally-selected individuals are not removed and the population size exceeds the maximum population size, the reaper queue mechanism removes the individual located at the top of the reaper queue (step 10-11) (Afterward this deletion is called *reaper deletion*). For example, when two offspring are added but only one individual (ind_2) is removed as shown in Fig. 2, one individual at the top of the reaper queue is additionally removed with the reaper deletion.

What should be noted here is that the fitness deletion probability P_d determines the ratio between the fitness deletion and the reaper deletion. In particular, when P_d is higher, the fitness deletion is increasingly executed, while the reaper deletion is decreasingly executed. On the other hand, when P_d is lower, the fitness deletion is decreasingly executed, while the reaper deletion is increasingly executed. Since a lot of individuals are removed before their evaluations completes if the reaper deletion increases, P_d can control how long the reaper deletion waits for the individuals that require long evaluation time.

4 Experiment

4.1 Settings

To investigate the effectiveness of ARE, we apply ARE to Linear GP (LGP) [8][9] using the machine code which we used in our previous research, and conduct experiments to compare GP using ARE (ARE-GP) with steady-state GP (SSGP) [7] as the asynchronous GP and TAGP as the asynchronous GP. The reason why we employ LGP is that an individual (program) in LGP has variable length chromosome and evaluation time of each individual generally differ from each other. Furthermore, since a machine-code program has probability to include the loop structure, individuals that include the infinite loop and do not complete their evaluation can be generated.

Our machine-code programs use the instruction set of the PIC10 [10] embedded processor developed by Microchip Technology Inc. It has 33, 12 bit instructions. These include addition, subtraction, Boolean logic, bitwise, and branch instructions. It does not include multiplication. For this reason, multiplication has to be achieved by repeating addition and bitwise operations in loop structures. A program can use any of 16 general purpose registers and one register (named working register). Each register consists of 32bits.

Test problems in this experiment are shown in Table 1. The problems are classified two types, one is Arithmetic problems that requires numeric calculations, and another is Boolean problem that requires logical calculation. In particular since Arithmetic problems require multiplication that is achieved with loop structures, they has high probability to generate programs that include infinite loops. In this experiment, the aim is to evolve program that minimize the time taken by the execution step by starting from an initial program that completely accomplishes the given task.

Table 1. Test problems in this experiment

Arithmetic		#data	Boolean		#data
A1	$f(x) = x^4 + x^3 + x^2 + x$	16	B1-2	{5,8}bit-Parity	{32,256}
A2	$f(x) = x^5 - 2x^3 + x$	16	B3-4	{5,7}bit-DigitalAdder	{32,128}
A3	$f(x) = x^6 - 2x^4 + x^2$	16	B5	6bit-Multiplexer	64
A4	$f(x, y) = x^y$	25	B6	7bit-Majority	128

Table 2. Parameters

Parameter	value	Parameter	value
#evaluations	10^6	Crossover rate	0.7
Max. program size	256	Mutation rate	0.1
Pop. size	100	Insertion rate	0.1
f_{max}	100	Deletion rate	0.1

The following fitness functions are respectively employed for Arithmetic and for Boolean:

$$f_{arith} = f_{max} - \frac{1}{n}\sum_{i=1}^{n} |\hat{y}_i - y_i^*| \tag{1}$$

$$f_{bool} = f_{max} - \frac{2}{n}\sum_{i=1}^{n} \delta(\hat{y}_i, y_i^*), \delta(x, y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases}, \tag{2}$$

where $\hat{y}_i$ indicates the i^{th} output value of a program, while y_i^* indicates the i^{th} target value. Note that the reason why the sum of difference is subtracted from f_{max} is that TAGP employs the parent selection depending on the accumulated fitness. Although ARE-GP and SSGP do not require such transformation, this experiment uses same fitness function. Individuals are compared in order of (1) fitness, (2) execution step, and (3) program size.

Common parameters in all GPs are shown in Table 2. The crossover combines two programs at two different crossover point, while the mutation randomly changes one random instruction in a program. The instruction insertion inserts one random instruction into random point, while the instruction deletion remove one random instruction from a program. In SSGP, the maximum execution step is set to $50,000$, and if a program does not complete in this limit, its fitness is evaluated as $-\infty$.

All experiments start from filling the population with an initial program. Each experiment is conducted 30 independent trials, and we evaluate GPs regarding the average execution step after the maximum number of evaluations.

4.2 Results

Table 3 shows the average execution step of the best program in the population after the maximum number of evaluations. In Table 3, all results are normalized by the average execution step of TAGP, i.e., the result of TAGP is 1 in

Table 3. Averages of the minimum execution step after the maximum evaluations (normalized by the result of TAGP). ARE-GP changes the archive size as {5, 10, 20, 30}.

Problem	SSGP	ARE-GP			
		archive size			
		5	10	20	30
A1	1.415	**0.852**	0.860	0.960	1.018
A2	1.429	**0.881**	0.883	0.964	1.120
A3	1.463	**0.863**	0.934	1.026	1.173
A4	0.976	0.881	**0.872**	0.919	0.903
B1	**0.954**	0.974	0.956	0.993	1.019
B2	1.007	0.991	**0.979**	0.986	1.023
B3	1.055	0.972	**0.967**	0.985	0.986
B4	1.024	0.965	**0.953**	0.965	0.968
B5	1.582	**0.864**	0.901	0.924	0.918
B6	1.174	0.891	0.909	**0.820**	0.990

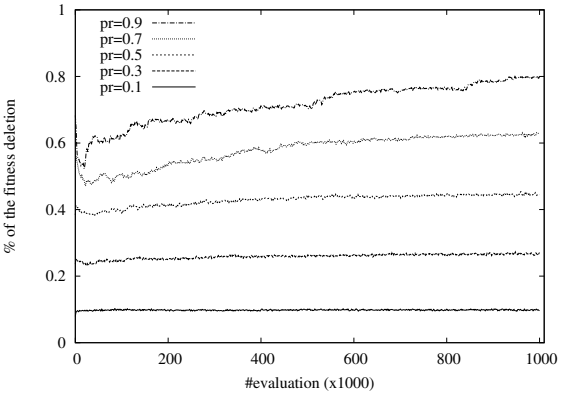
Table 4. Averages of the minimum execution step after the maximum evaluations (normalized by the result of TAGP). ARE-GP changes the fitness deletion probability P_d as {0.1, 0.3, 0.5, 0.7, 0.9}, and the archive size is 5 ($P_d = 0.5$ is the same as Table. 3).

Problem	ARE-GP				
	fitness deletion probability P_d				
	0.1	0.3	0.5	0.7	0.9
A1	2.273	0.880	0.852	**0.817**	0.868
A2	2.401	0.881	0.881	0.863	**0.836**
A3	2.607	0.886	0.863	**0.862**	0.880
A4	3.615	3.615	**0.881**	0.931	1.132
B1	0.974	0.976	0.974	0.954	**0.942**
B2	1.018	0.973	0.991	**0.956**	0.982
B3	0.977	0.973	**0.972**	0.988	0.979
B4	0.965	0.958	0.965	0.963	**0.955**
B5	**0.857**	0.883	0.864	0.887	0.891
B6	**0.890**	1.025	0.891	0.965	0.917

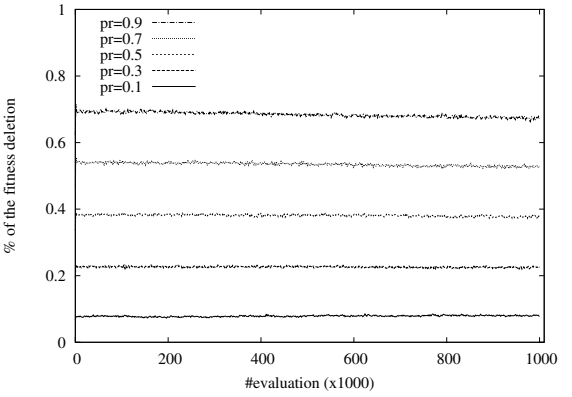
all problems, and the best result (the shortest execution step) in each problem is indicated as **bold** style. In ARE-GP, we confirm different archive sizes $5, 10, 20$, and 30. From these results, it is easily confirmed that ARE-GP can asynchronously evolve programs using only relative evaluation and without the thresholds required in TAGP. Furthermore, it is revealed that ARE-GP outperforms TAGP in all problems. In particular small archive size such as 5 and 10 reliably gives better performance. The reason why large archive size such as 30 is not good is that since ARE avoids overlap of programs in the archive, low fitness programs remain in the archive and the selection pressure depending on the reference program decreases. From this fact, it is indicated that the archive size should be set as small size.

To verify the effect of the difference of the fitness deletion probability P_d, we confirm different probability $0.1, 0.3, 0.5, 0.7$, and 0.9. Here the archive size is set as 5, and $P_d = 0.5$ results the same as the previous one. The results are shown in Table 4, where all results are also normalized by the result of TAGP, and the best result in each problem is indicated as **bold** style.

From these results, it is revealed that high fitness deletion probability such as $P_d \geq 0.5$ is effective in most problems, while, in all Arithmetic problems,



(a) Arithmetic 1



(b) Boolean 1

Fig. 3. Percentage of the fitness deletion, where the archive size is 5, and $P_d = 0.5$

small probability $P_d = 0.1$ is worse than the results of TAGP. Actually, these cases hardly evolve programs from the initial program. Arithmetic problems requires loop structures to achieve multiplication, in particular Arithmetic 2 include double loops to calculate x^y. If the loop structures are broken with the genetic operation, offspring lose the loops and they quickly complete their evaluation. Such programs succumb to reaper deletion so most programs that include loops are removed before their evaluation. To wait loop calculation and to remove programs that are evaluated to quickly, we have found a high fitness deletion probability to be effective.

As mentioned above, the fitness deletion probability P_d determines the ratio between the fitness deletion and the reaper deletion. Fig. 3 shows the change of the ratio of the fitness deletion that is calculated as $\%_{cp} = \#del_{cp}/(\#del_{cp} + \#del_{rp})$, where p_{cp} indicates the percentage of the fitness deletion, while $\#del_{cp}$ and $\#del_{rp}$ respectively indicate the number of the fitness deletion and the reaper

deletion in a certain evaluations. In Fig. 3, the horizontal axes show the number of evaluations, while the vertical axes show the percentage of the fitness deletion. Each line shows the average $\%_{cp}$ in the case that $P_d = \{0.1, 0.3, 0.5, 0.7, 0.9\}$. Note that although these figures only show the results of Arithmetic 1 and Boolean 1, we confirm same trend in all other problems in each problem type.

From these results, the fitness deletion probability can controls the ratio of two deletions. In particular, when the fitness deletion probability is low, the ratio of the fitness deletion is low, while it is high, the ratio of the fitness deletion is high. This indicates that ARE can consider how long the reaper queue deletion waits for the individuals that require huge computational time by changing the fitness deletion probability. In Arithmetic problems, the percentage of the fitness deletion is lower in the early stage of evolution even if the fitness deletion probability is high. At this stage, programs that are better than the archive is easily evolved, so the replacement between a temporally-selected program and an archived program often occurs. For this reason, even if the fitness deletion probability is high, the reaper deletion is executed in the early evolution because of decreasing the fitness deletion.

5 Conclusion

This paper proposed a novel *asynchronous reference-based evaluation* (named as ARE) for an asynchronous EA that evolves individuals independently unlike general EAs that evolve all individuals at the same time. ARE is designed for an asynchronous evolution by tertiary parent selection and its archive. In particular, ARE asynchronously evolve individuals through a comparison with only three of individuals (i.e., two parents and one *reference* individual as the tertiary parent) unlike synchronous evolution which involves a comparison with all population members. ARE improves its performance by archiving good individuals as the *reference* individual. To investigate the effectiveness of ARE, this paper applies it to the Linear GP (LGP) problems. We have conducted experiments that aim to minimize the execution step of machine-code programs. An experiment comparison of ARE-GP with steady-state GP (SSGP) as the synchronous GP and our previous GP (Tierra-based asynchronous GP: TAGP) as the asynchronous GP, produced the following implications: (1) ARE-GP asynchronously successfully evolved machine-code programs, showing that ARE-GP does not require the thresholds of TAGP, (2) ARE-GP outperformed TAGP in all test problems, in particular, smaller archive size in ARE-GP reliably gives better performance

What should be noticed here is that these results have only been obtained from one type of problem, i.e., Linear GP. Therefore, further careful qualifications and justification, such as an analysis of results using other general LGP problems such as symbolic regression or classification problem, are needed to extend the range of application of ARE to other EA domain. Such important directions must be pursued in the near future in addition to the following future research: (1) the parallelization under the ARE framework because the asynchronous approach is suitable for the parallelization; and (2) an adaptation of the fitness deletion rate

P_d and the archive size depending on the evolution degree or the diversity of the population because these parameters gives a big influence to the performance of ARE.

Acknowledgments. This work was supported by Grant-in-Aid for JSPS Fellows Grant Number 249376.

References

1. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
2. Koza, J.: Genetic Programming On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
3. Lewis, A., Mostaghim, S., Scriven, I.: Asynchronous multi-objective optimisation in unreliable distributed environments. In: Lewis, A., Mostaghim, S., Randall, M. (eds.) Biologically-Inspired Optimisation Methods. SCI, vol. 210, pp. 51–78. Springer, Heidelberg (2009)
4. Glasmachers, T.: A natural evolution strategy with asynchronous strategy updates. In: Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference, GECCO 2013, pp. 431–438. ACM, New York (2013)
5. Harada, T., Takadama, K.: Asynchronous evaluation based genetic programming: Comparison of asynchronous and synchronous evaluation and its analysis. In: Krawiec, K., Moraglio, A., Hu, T., Etaner-Uyar, A.Ş., Hu, B. (eds.) EuroGP 2013. LNCS, vol. 7831, pp. 241–252. Springer, Heidelberg (2013)
6. Ray, T.S.: An approach to the synthesis of life. Artificial Life II XI, 371–408 (1991)
7. Reynolds, C.W.: An evolved, vision-based behavioral model of coordinated group motion. In: Proc. 2nd International Conf. on Simulation of Adaptive Behavior, pp. 384–392. MIT Press (1993)
8. Banzhaf, W., Francone, F.D., Keller, R.E., Nordin, P.: Genetic programming: an introduction: on the automatic evolution of computer programs and its applications. Morgan Kaufmann Publishers Inc., San Francisco (1998)
9. Brameier, M.F., Banzhaf, W.: Linear Genetic Programming, vol. 117. Springer (2007)
10. Microchip Technology Inc.: PIC10F200/202/204/206 Data Sheet 6-Pin, 8-bit Flash Microcontrollers. Microchip Technology Inc. (2007)