
Proceedings Paper:
Ma, Yunfeng and Indrusiak, Leandro Soares (2016) Hardware-accelerated parallel genetic algorithm for fitness functions with variable execution times. In: GECCO 2016 - Proceedings of the 2016 Genetic and Evolutionary Computation Conference. 2016 Genetic and Evolutionary Computation Conference, GECCO 2016, 20-24 Jul 2016 ACM, USA, pp. 829-836.

https://doi.org/10.1145/2908812.2908879

Hardware-Accelerated Parallel Genetic Algorithm for Fitness Functions with Variable Execution Times

Yunfeng Ma
Department of Computer Science
University of York
Deramore Lane, YO10 5GH, UK
ym608@york.ac.uk

Leandro Soares Indrusiak
Department of Computer Science
University of York
Deramore Lane, YO10 5GH, UK
leandro.indrusiak@york.ac.uk

ABSTRACT

Genetic Algorithms (GAs) following a parallel master-slave architecture can be effectively used to reduce searching time when fitness functions have fixed execution time. This paper presents a parallel GA architecture along with two accelerated GA operators to enhance the performance of master-slave GAs, specially when considering fitness functions with variable execution times. We explore the performance of the proposed approach, and analyse its effectiveness against the state-of-the-art. The results show a significant improvement in search times and fitness function utilisation, thus potentially enabling the use of this approach as a faster searching tool for timing-sensitive optimisation processes such as those found in dynamic real-time systems.

CCS Concepts

•Computer systems organization → Parallel architectures; •Hardware → Statistical timing analysis;

Keywords

Genetic algorithms; Hardware realization; Parallelization; Speedup technique; Time-tabling and scheduling.

1. INTRODUCTION

The search efficiency of a Genetic Algorithm (GA) can determine the range of problems which it can address. The Master-Slave model [3] has been widely used to parallelise GAs, and is able to achieve significant performance improvements particularly when its fitness function has fixed execution time.

However, not all problems can be abstracted as a fitness function with fixed running time. Tree search, address comparison and schedulability analysis [12] are good examples of algorithms with input-dependent execution times that have been used as fitness functions in GAs. The use of such fitness functions within a Master-Slave GA can still yield performance benefits, but at a much lower efficiency level since slave fitness functions run in lock-step and have to wait for the slowest one to finish before starting another evaluation. The level of inefficiency increases with the magnitude of variability of the fitness functions' execution times, and with the actual number of slave fitness functions operating in parallel.

This paper explores extensions to the Master-Slave GA architecture, aiming to increase search efficiency when using fitness functions with variable execution time. We propose a novel hardware-accelerated parallel GA which overcomes the lock-step nature of the Master-Slave model, and evaluate its advantages in the cases of fitness functions with large, moderate and no variability in their execution times. In addition, we take advantage of the proposed hardware-based platform and introduce an approach to hardware-accelerate and pipeline crossover and mutation operators. We then implement in hardware a comparable GA based on the original Master-Slave architecture and use it as a baseline to highlight the advantages of the proposed approach.

The paper is organised as follows: Section 2 reviews related work and is followed by a problem statement in Section 3. The proposed hardware architecture and implementation are presented in Section 4; the experimentation platform and results analysis are described in Section 5, followed by our conclusions.

2. RELATED WORK

2.1 Architecture of GA

Since a GA fitness function can be a computationally-complex algorithm, and since it is launched many times during the GA searching process, it is usually the most time-consuming process in a sequential GA. To alleviate this problem, researchers proposed a global parallel GA (named as Master-Slave model) [3] to launch a number of fitness functions simultaneously. It can accelerate the candidates evaluation and thus reduce the searching time, compared with sequential GA. Later the authors of [3] also suggested two others GA models (a semi-synchronous and a distributed asynchronous concurrent) for further improving the Master-Slave model GA. This can also be seen in [4] and [5].

2.2 Implementation of GA

Decades of research have also improved GA performance from an implementation point of view as well. The authors of [6] presented a hardware implementation of a sequential GA, which they further refined in [7]. Although they applied a parallel parent selection, the performance improvement is not very significant and its memory interference component

ACM ISBN 0000.
DOI: 000

is rather difficult to implement.

In addition, the researchers also proposed various GA operators to reduce the searching time of GA. The authors in [8] introduced an implementation of GA operators for compact GA which is suitable for binary coding style. In [10], [9] and [11], the authors presented implementation for either both crossover and mutation or only crossover to make improvement in hardware.

2.3 Fitness Categories

GAs have been used with a wide variety of fitness functions. According to their execution time, these function functions can be divided into two classes. The fitness functions in the first class have fixed execution time no matter what the inputs are. The second class consists of the fitness functions that have variable execution time. However, the variability of running time may be considerable. Some fitness functions have moderate variable execution time such as End-to-End Response Time Analysis (E2ERTA) which is used to analyse the timing performance of real-time Networks-on-Chip (NoC) [12]. One candidate's execution time can be only a few times higher than the others'. However, there are fitness functions with significant variability such as network address checking and tree searching problem, where the running times can differ by more than one order of magnitude.

3. PROBLEM STATEMENT

In the following sections, the state-of-the-art Master-Slave GA is referred as Lock-Step Master-Slave GA (LS-MS GA) and is used as a baseline in the comparative analysis of the proposed approach. In each generation, its searching time can be divided into GA operation time (by the GA operators themselves) and candidates evaluation time (by the fitness function). We will address two of its shortcomings, namely the lock-step problem and implementation limitations, which are made more severe when applied with fitness functions with variable execution time.

3.1 Lock-step problem

Assuming a LS-MS GA's population size is 4 and the number of fitness functions is 2, its architecture has been shown in Fig. 1a. Whenever the master is ready (all candidate solutions have already been evaluated) and all fitness functions are in idle state, the master will assign two chromosomes to fitness function 0 and 1 respectively and launch them simultaneously. The second round release will only be started when all results have already been collected by the master in order and all fitness functions have been in idle state again.

This architecture will not affect the execution time of candidates evaluation when the computation time of all fitness functions are same and fixed as it has been shown in Fig. 1b. However, it will suffer a lock-step problem and produce a significant adverse impact to the evaluation time if the fitness functions' computation time are variable and depend on different candidate solutions. Fig. 1c shows this phenomenon. It can be seen that the fitness 1 can only store its results after fitness 0 has been finished and the results have been recorded, no matter how fast fitness 1 can be executed. This problem can become much worse when the size of GA population, the number of fitness functions and the variability of fitness function execution time is increased.

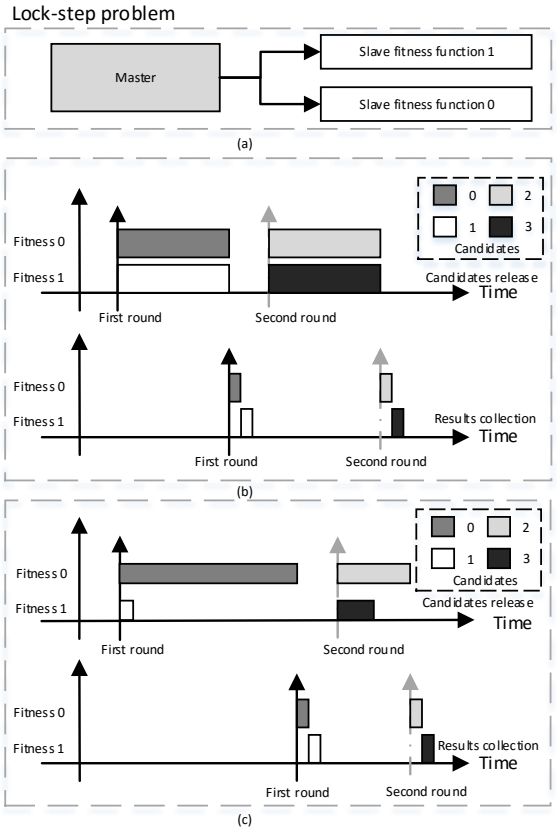

Figure 1: (a) Lock-step Master-slave GA Architecture, (b) Fitness Function with Fixed Execution Time, (c) Fitness Function with Variable Execution Time

Note: Population size is 4.
Number of fitness functions is 2.

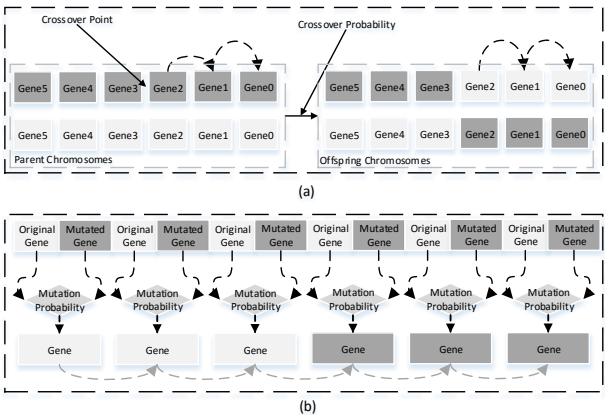

Figure 2: (a) Software Limitation of Crossover Operation, (b) Software Limitation of Mutation Operation

3.2 Implementation limitation

The LS-MS GA can also suffer implementation limitations. Most state-of-the-art implementations of GA are based on software running on regular CPUs. Although researchers can apply distributed computation to execute multiple fitness functions simultaneously, the GA operators cannot be

well supported by a software implementation based on typical CPUs, because of the low efficiency in processing vectors and lack of pipeline structure. This can be seen on the processing of crossover and mutation operators, detailed below.

3.2.1 Low Efficiency in Processing Vector

Crossover operations require the swap of selected parts of two parent chromosomes (if the crossover condition has been satisfied), as shown in Fig. 2a. Since chromosomes are normally stored as arrays in software, software inevitably will swap these two arrays element by element, which has been presented by black dash arrows. Such operations will invariably take several clock cycles of a typical CPU, even in the case of partial swaps. The number of clock cycles will be further increased with the size of chromosome.

Mutations can suffer similar limitations in software implementation. Its working process requires to check each gene whether it should be mutated or not and generate a mutated value if needed. An example has been shown in Fig. 2b. This process also requires multiple clock cycles of a typical CPU, also scaling up that time with the increase of the number of genes.

3.2.2 Lack of Pipeline Structure

Besides, software cannot provide a pipelined structure. This can be seen from the working procedure of reproduction (produce offspring chromosomes by using crossover and mutation). Its working procedure can be either applying crossover over the whole parent population first and then using mutation to generate the final offspring population, or each time applying crossover and mutation sequentially only over two selected parent chromosomes and repeat this process until the whole offspring population has been generated. These two kinds of procedures have been illustrated in Algorithm. 1.

Algorithm 1 Reproduction Working Process

```
1: procedure TYPE 1
2:     for Number of offsprings < Population Size do
3:         Select parent chromosomes
4:         Crossover
5:     for Number of offsprings < Population Size do
6:         Mutation
7: End Procedure
```

```
1: procedure TYPE 2
2:     for Number of offsprings < Population Size do
3:         Select parent chromosomes
4:         Crossover
5:         Mutation
6: End Procedure
```

No matter which type we will use, when one operator is executing, the other one has to be paused, this phenomenon has been shown in Fig. 3a and b. This will increase the computation time compared with a pipelined structure whose timing difference has been presented in Fig. 3c.

It can be seen that reproducing the first offspring chromosome costs the same time used by the two procedures in Algorithm 1. However, after the first offspring chromosome, in each stage there will be one new candidate chromosome generated. If we assume the number of clock cycles used

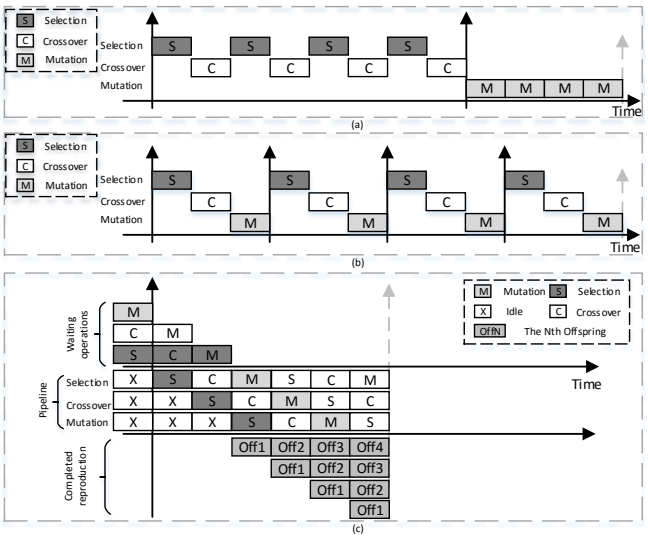

Figure 3: (a) Type 1 runing time, (b) Type 2 runing time, (c) Pipelined structure running time

Note: Following example of Fig. 1.

by selection, crossover and mutation are same and equal to N and the population size follows the example in Fig. 1, the total numbers of clock cycles to finish reproduction can be represented by $6 * 2 * N$ for both Type 1 and 2 of Algorithm 1. The total number used by pipelined structure should be $(3+3) * N$. Thus, pipelined structure can be used to improve the timing performance of reproduction and this improvement will be significant when the population size increased.

4. ARCHITECTURE DESCRIPTION

As analysed in previous section, the state-of-the-art LS-MS GA cannot efficiently reduce the searching time when applying a fitness function whose execution time is variable and depends on candidate solutions. In order to alleviate this problem, we discuss the possibility of using an improved architecture (Free-Step Master-Slave model). In addition, we also introduce two hardware GA operators to enhance GA's timing performance and followed by a description of our implementation in this section.

4.1 Free-Step Master-Slave model

Based on the characteristic of lock-step problem, we can find that this problem is caused by the system synchronization. In each release round, the master has to synchronise the data for both fitness execution and results collection. This blocks the further step of idle fitness functions to get unevaluated candidate solutions when other fitness functions are still executing. Therefore, we proposed a possible asynchronous model to solve this problem. One example has been shown in Fig. 4.

This example follows the one in Fig. 1c. The two light gray solid arrows indicate the timing points of finishing executing all fitness functions and collecting all results of Fig. 1c respectively. From Fig. 4, we can see that if we can load, release fitness and collect result individually, the over-

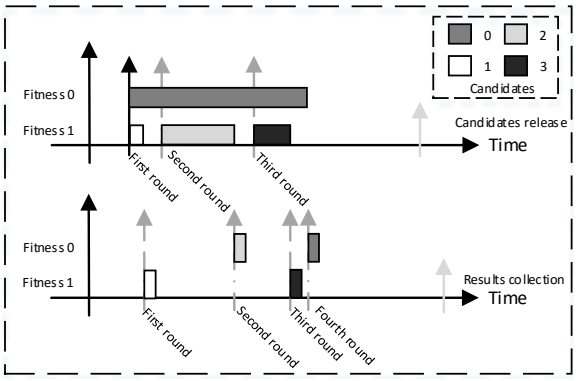

Figure 4: Example of Free-Step Master-Slave GA

Note: Follows the example in Fig. 1c.

all execution time of candidate evaluation can be reduced significantly.

4.2 Accelerated GA Operators

As aforementioned, the limitations of software implementation are the bottle-neck we need to solve. Chromosomes treated as arrays in software can be represented as vectors and easily operated by hardware. Therefore, we propose an implementation in hardware to accelerate the crossover and mutation operators. We also assembled them with a pipeline structure to further reduce the timing cost.

4.2.1 Crossover

For crossover operator, we introduced a Crossover Mask and several logic gates ('NOT', 'AND' and 'OR') to swap two parent chromosomes according to crossover point and possibility. The Crossover Mask is a series pre-designed vectors determined by both the number of gene and the width of gene (the number of bits to represent one gene). An example of how the proposed crossover works has been shown in Fig. 5a where each gene is represented by 1-bit and the total number of gene is 10.

The crossover point (randomly generated by a Random Number Generator) will be used as the index of a look-up table which stores the 'Crossover Masks'. The two parent chromosomes will be transferred through two logic 'AND' gates with Crossover Mask and NOT Crossover Mask respectively. The results of logic 'AND' gates will be applied as the inputs of a logic 'OR' gate to generate the potential crossover result. Whether the final crossover result should be the swapped result (potential crossover result) or the original parent chromosome is determined by a comparison between the pre-configured crossover rate and a given crossover probability which is also randomly generated by a Random Number Generator. If the crossover condition has been satisfied, the final crossover result will be the swapped result, otherwise the parent chromosome will be selected.

Since the propagation delay among ROM and combinational logic are slightly. Therefore, when the frequency of the whole system is not extremely high, the proposed crossover operation can be finished within one clock cycle no matter where the crossover point is and how many genes there are. For an extremely high frequency system, we can introduce pipeline structure to break down this working process, in order to make this operation can be finished within one clock cycle.

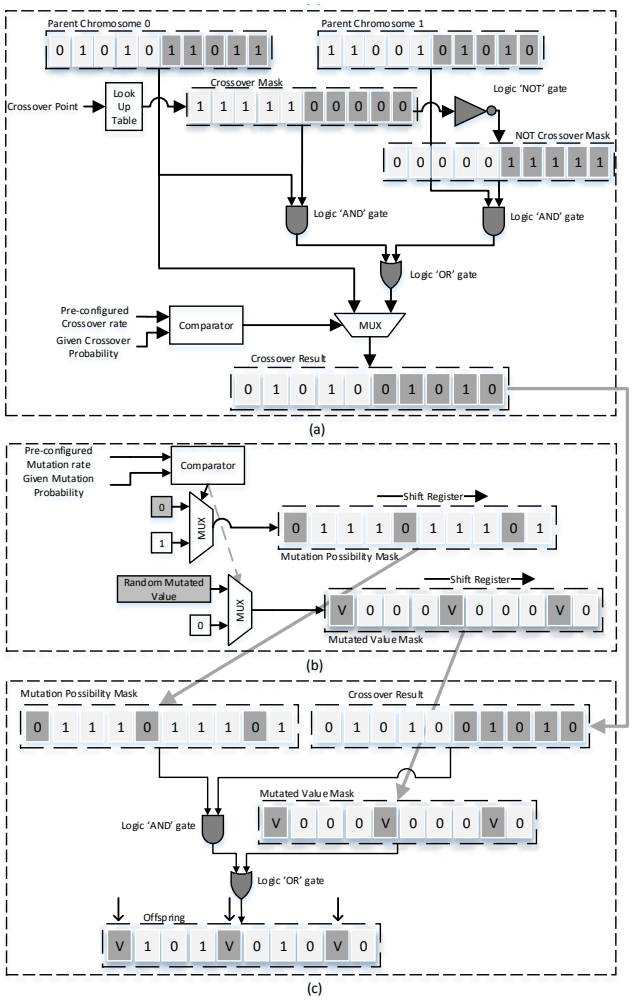

Figure 5: a. Crossover Strategy Example. b. Mutation Template Generator. c. Mutating component Example.

Note: Number of gene is 10. Width of gene is 1.
The input vectors of Fig. 5c are from both Fig. 5a and 5b.
The "V" represents the mutated value.

4.2.2 Mutation

Similar to crossover, the idea of accelerating the mutation operator is also based on mask vectors. An example of mutation working process has been illustrated by Fig. 5b and c. It can be seen that, the proposed mutation operator can be divided into Mutation Template Generator and Mutating component.

Mutation Template Generator.

In Fig. 5b, the Mutation Template Generator consists of a Mutation Possibility Mask and a Mutated Value Mask. They are used to determine whether each gene of a chromosome should be mutated and provide the mutated values when needed. Their generating procedure can be described

as follows. In each clock cycle, a given mutation probability (randomly generated) will be compared with the preconfigured mutation rate (similar with crossover rate). The result of this comparison will be used to indicate whether the current gene should be mutated. If the current gene needed to be changed, one bit logic '0' will be shifted into the Mutation Possibility Mask and a random generated mutated value will be shifted into the Mutated Value Mask. Otherwise, if the current gene should be maintained, one bit logic '1' and a logic '0' vector (all bits are logic '0' if a gene represented by multiple bits) will be shifted into the Mutation Possibility Mask and the Mutated Value Mask respectively. This process will be repeated until all genes of a chromosome has been checked.

Mutating component.

The mutating component will read the mutation template. In its working process, shown in Fig. 5c, the crossover result will be transferred to a logic 'AND' gate with the Mutation Possibility Mask. Their result will be applied as one input of a logic 'OR' gate to calculated the final offspring with the Mutated Value Mask. Similar with crossover operator, all the operation in this mutating component are combinational logic, the delay among them is only propagation delay. Therefore, the mutation operator can be finished within one clock cycle no matter how many genes a chromosome has.

Further Optimisation.

We can simply notice that by using this generator, only one gene's template can be generated within one clock cycle. However, it does not mean this idea cannot be used to accelerate the mutation operator. In order to solve this problem, we introduced a template FIFO (First-In-First-Out) to store mutation templates before the mutation operator is executed. Since one of the natural characteristics of hardware is parallel computing, we can easily launch the Mutation Template Generator to produce and store templates when GA is in other stage such as candidate evaluation. In addition, some chromosomes' templates can be generated when the mutation operator is executing. To proof this, we can make an assumption that:

- the Population Size is m;
- the Number of Gene is n;
- the Minimum Depth of FIFO is x.

We can know that m clock cycles are required to finish the mutation operation over the whole population, n clock cycles are needed to generate one template. Currently only x chromosomes' templates are ready in FIFO. The worst case situation can be that the mutation operation and the Mutation Template Generator (to generate the rest templates) are released at the same point. In other words, we have to prepared the remaining templates within m clock cycles. Thus, we can get Equation 1a and b. Therefore, we can minimise depth of FIFO to $\frac{(n-1)m}{n}$ to reduce the hardware resource cost.

$$m \geq (m - x) * n \tag{1a}$$

$$x \geq \frac{(n-1)m}{n} \tag{1b}$$

4.2.3 Reproduction Pipeline

As mentioned above, the software implementation lacks in pipelined structure. To solve this limitation, we assembled the proposed crossover and mutation operator with a possible pipeline style with additional registers which has been presented in Fig. 6. By applying this architecture, the first two clock cycles will generate two invalid offspring. However, after that there will be two valid offspring reproduced every clock cycle. Therefore, the execution time of reproduction can be reduced significantly.

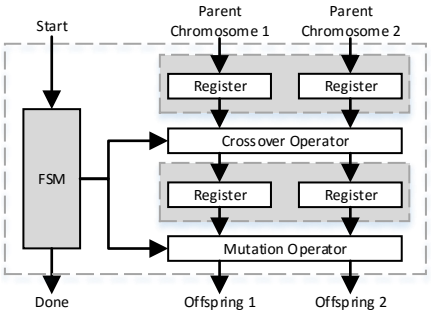

Figure 6: Reproduction Pipeline Architecture.

4.3 Hardware Free-Step Master-Slave GA

We combined the proposed model (Free-Step Master-Slave model) and the accelerated GA operators in a top-level hardware architecture (FS-MS GA) which has been shown in Fig. 7 as our proposed implementation.

The working process of hardware FS-MS GA is similar to the one of LS-MS GA. In order to launch fitness functions and collect feedback asynchronously, we introduced several components such as Intermediary Register Bank (IRB), Arbitration, Fitness Function ID Coder (FFID-Coder) and Combined Population Register Bank (CPRB). The IRB is used to temporarily store the new candidate solutions which can be generated randomly through initialization or breeded by crossover and mutation in reproduction component, if these solutions cannot be evaluated immediately. Whenever new chromosomes arrive in IRB, the Arbitration will distribute them to fitness functions according to the indication from FFID-Coder. The FFID-Coder collects the busy and done signals from each fitness function. It generates two address signals for both Arbitration and CPRB to support candidate distributing and results storing respectively. Serial written feedback signals will also be generated by FFID-Coder to fitness functions. The CPRB will store both parent and offspring populations. Its size is twice of the parent population's size. The replacement will sort the CPRB according to a given strategy such as ranking (the better a solution is, the lower address it will be put). The selector will generate two addresses to pop out two parent chromosomes from parent population for reproduction according to selection strategy.

4.3.1 Arbitration and Fitness Function ID Coder

Distributing candidate chromosomes to each fitness function asynchronously is achieved by the Arbitration and FFID-Coder. The Arbitration is triggered by coded fitness ready address and IRB ready signals. Its architecture has been shown in Fig. 8a. In its working process, the 'coded fitness ready address' can indicate whether there are fitness

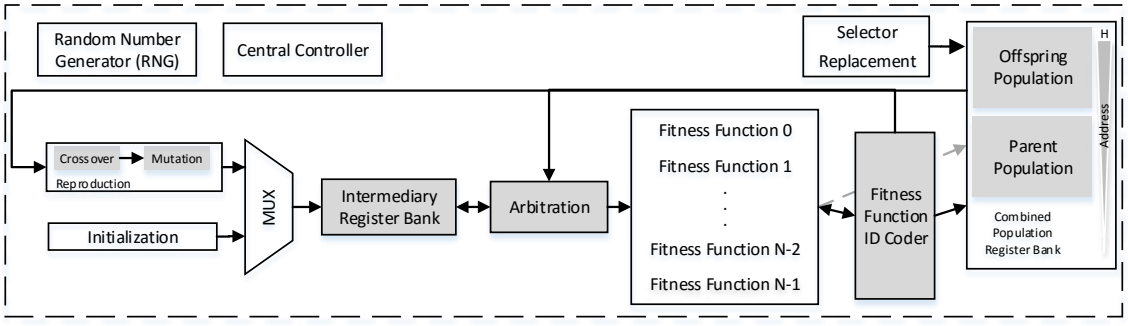

Figure 7: Hardware FS-MS GA Architecture

functions ready to receive new candidate chromosomes. If there are and the IRB ready signal is valid (there is at least one candidate chromosome in IRB has not been evaluated), the Arbitration will enable the 'read enable' signal to read one chromosome from IRB and distribute it to the right slot of chromosome vector according to 'coded fitness ready address'. Otherwise, both the 'read enable' signal and chromosome will be disabled by logic '0' and 'Zero Vector' respectively.

The FFID-coder shown in Fig. 8b collects the ready and done signals from each fitness function. The ready signals are used to geneate a fitness function ready address which can guide the chromosome distribution in Arbitration. The done signals will be encoded to indicate CPRB to read result from which fitness function. Whenever the result has been recorded, the related bit in written vector will be set as the acknowledgement back to the fitness function. If there are more than one fitness functions idle or finished, the FFID-coder will code based on priority of fitness functions. If there is no fitness function idle or finished, the FFID-coder will set all output signals invalid.

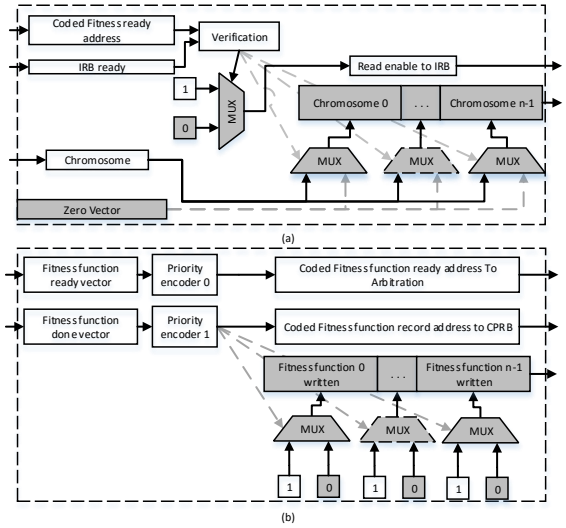

Figure 8: a. Arbitration Architecture. b.Fitness Function ID Coder Architecture.

Note: 'Zero Vector' consists of logic '0'.
IRB refers to Intermediary Register Bank.
CPRB refers to Combined Population Register Bank.

5. EXPERIMENTATION PLATFORM AND RESULTS ANALYSIS

In this section, testing fitness functions, experimentation platform, experimentation configuration and results analysis will be discussed to show the performance of our proposed implementation.

5.1 Fitness Functions

In order to evaluate how well HW FS-MS GA can improve over HW LS-MS GA with various kinds of fitness functions, we propose three implementations of fitness functions with different levels of variability of their execution time.

5.1.1 Fixed Excution Time

The first fitness function is a Logic One Counter (LOC) which returns the number of logic '1's of an input vector. Its execution time is fixed and depended on the size of input vector only since its finish condition is when all bits of the input vector have been checked.

5.1.2 Moderate Variable Excution Time

The moderate variation requires the magnitude of the fitness function's variable execution time to be not significant (means one's running time is times or many times of the other one's). E2ERTA which can be used to analysis NoC's hard real-time timing performance is a choice to represent this kind of fitness functions. Although, the execution time of E2ERTA can be different, depending on variable input (task allocation or mapping of NoC), the magnitudes of variations are slight. In addition, its input can be abstracted as a chromosome and optimised by GA. An example can be seen from [2]. Moreover, a hardware version of E2ERTA presented in [1] can be directly used in our HW-FSMS GA experimentation. Therefore, E2ERTA is a realistic example of a fitness function with moderate variable execution time.

5.1.3 Large Variable Excution Time

The large variable execution time can be imitated by Slice Logic One Counter (S-LOC). The input of S-LOC consists of a slice range and a test vector. It can return the number of Logic '1's in a slice of an test vector. A slice can cover from 1-bit to the whole test vector. Its range is represented by the exponent of a given base. Therefore, the width of a slice can be represented by Equation 2.

$$Slice = Test\ Vector\ [Base^{SliceRange+1} - 1\ down\ to\ 0] \quad (2)$$

Since the finish condition of S-LOC is when all bits in the slice have been checked, the variation of its execution time can be significant.

5.2 Experimentation Platform

To evaluate the performance of FS-MS GA in hardware, we propose an experimentation platform which is an embedded system based on Xilinx VC709 develop board shown in Fig. 9a. We packet our hardware implementations as customer peripherals and mount them on an AXI bus which is an on-chip interconnect link used in Xilinx system-on-chip design. The FS-MS GA and LS-MS GA simultaneously with either LOC or S-LOC as their fitness function, since the resources cost of LOC and S-LOC are low. However, because the E2ERTA is resource costed, only one GA model (LS-MS or FS-MS) can be executed at one time.

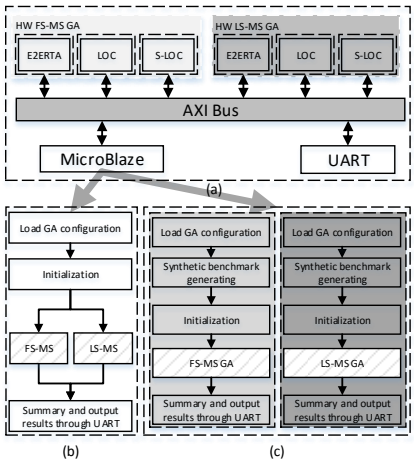

Figure 9: (a) Experimentation Platform, (b) Test for LOC and S-LOC, (c) Test for E2ERTA

The working process of these two testing are similar. The MicroBlaze firstly load GA configuration (number of fitness, crossover rate, mutation rate, size of population and so on) to testing components. Since E2ERTA requires task information and application information to compute results, an extra step (load synthetic benchmark [2]) has to be executed in E2ERTA testing. After, the hardware GA will initialize the population and continue the searching or evaluation, until the finish condition (either a number of generation has been analysed or at least one suitable candidate has been found) has been achieved. Then, the MicroBlaze will collect data from hardware components, and organise these results. The results are output through a UART port.

5.3 Experimentation Configuration

To measure the performance of our proposed implementation in various situations, we configured our experimentations as follows:

- Crossover rate is 0.5%;
- Mutation rate is 0.01%;
- Population size is 6, 8, 16;
- Number of fitness is 2, 3, 4, 5 when Population is 6;
- Number of fitness is 2, 4 when Population is 8 and 16.

Because GA is a stochastic searching, one time testing cannot illustrate the difference among these two models (LS-MS and FS-MS models). Therefore, we increase the number of testing times to 1000000, in order to have a better coverage.

5.4 Results Analysis

Since the proposed hardware includes two improvement directions (GA architecture and accelerated GA operators), we need to understand where the improvement is from. Therefore, we measure the number of clock cycles of GA operators and Fitness Functions execution respectively. The detailed results are shown in Table. 1 (No.FF refers to Number of Fitness Functions, the results in this table are average number of clock cycle used by each generation).

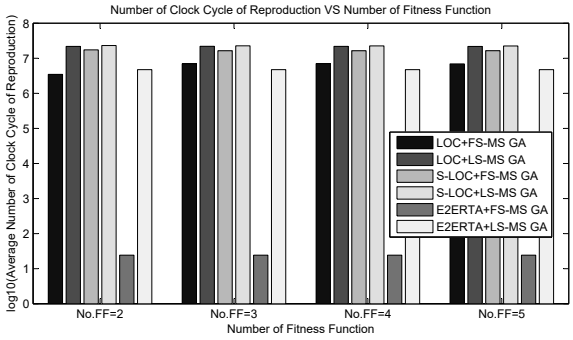

Figure 10: The Improvement of Reproduction

Note: Population size is 6.
No.FF refers as number of fitness functions.

5.4.1 *Improvement from GA Operators*

Figure. 10 shows the acceleration of reproduction under different type and number of fitness functions when the population size is fixed as 6. The Y-Axis shows the average numbers of clock cycles that has been used to finish a reproduction operation in one generation. Because the numbers are too large, we arrange them in log10 style.

It can be seen that for all situations the proposed accelerated reproduction will use much less number of clock cycles than LS-MS GA. However, for each evaluation method, increasing the No.FF cannot make a significant improvement when the population size is fixed. LOC's performance will suffer a slightly decrease by changing No.FF from 2 to 3 but it remains after. This is because LOC is a fixed and relatively slow evaluation method compared with S-LOC. When No.FF is 2, the speed of consuming mutation templates is not very fast. The pre-stored templates can support LOC for a while. However, when the No.FF has been increased to 3, the consuming speed has been raised. The mutation template FIFO will be soon out of stock and make the following operations paused for waiting new templates. Fortunately, there is a upper bound for this pause. This is why LOC can be maintained. For S-LOC, because it can be very fast in most situations, even 2 (No.FF) can already make it achieve its pause upper bound. For the slowest fitness

Table 1: Results Table

		LOC						S-LOC						E2ERTA					
		LS-MS		FS-MS		Improvement(%)		LS-MS		FS-MS		Improvement(%)		LS-MS		FS-MS		Improvement(%)	
PopSize	No.FF	RD	FF	RD	FF	RD	FF	RD	FF	RD	FF	RD	FF	RD	FF	RD	FF	RD	FF
6	2	1539	786	693	777	48.92	1.15	1576	107	1394	77	11.55	27.65	792	243840	4	228307	99.49	6.37
	3	1543	530	937	520	65.64	1.89	1564	97	1358	61	13.15	37.10	792	229760	4	214849	99.49	6.49
	4	1540	530	941	520	65.58	1.89	1561	101	1359	61	12.94	39.82	792	229540	4	214735	99.49	6.45
	5	1537	530	934	520	65.51	1.89	1558	102	1361	60	12.67	40.82	792	229180	4	213985	99.49	6.63
8	2	2179	1048	1104	1036	51.89	1.15	2097	173	1998	112	4.71	34.88	1060	500711	5	471670	99.53	5.80
	4	2196	536	1561	522	75.59	2.61	2194	142	1924	81	12.32	43.05	1059	251672	5	228971	99.53	9.02
16	2	4138	2096	1824	2072	49.34	1.15	4160	487	3653	296	12.21	39.09	2115	896560	9	870918	99.57	2.86
	4	4178	1072	1839	1056	74.34	1.49	4335	381	3779	118	12.83	69.00	2112	812150	9	760172	99.57	6.40

Note: PopSize is Population Size; No.FF is Number of Fitness Functions; RD is Reproduction; FF is Fitness Functions.

function (E2ERTA), its evaluation time is long enough for mutation template FIFO to refill. Therefore it will reach the lower bound of number of clock cycles.

If we fixed No.FF and increase the population size, the required number of clock cycles will be increased, however, it is still much better than LS-MS GA, which has been labelled by dark grey in Table. 1.

5.4.2 *Improvement from GA Architecture*

For LOC, although the proposed architecture can reduce the number of clock cycles used for fitness functions, the improvement is slightly. However, the S-LOC can be well enhanced which has been shown in Fig. 11. It can be seen the improvement of S-LOC is increased with the No.FF raised. Take the population size is 6 as an example. The improvement grows rapidly at the beginning and converge after. Therefore, this improvement has a upper bound which is determined by the variability and number of fitness functions.

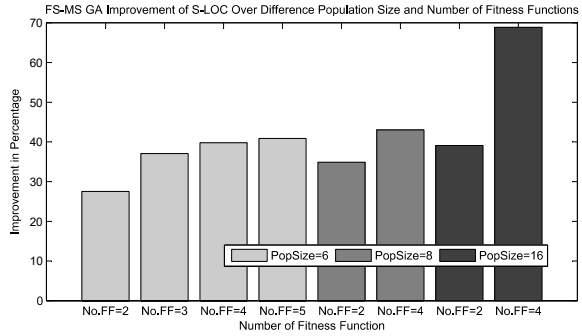

Figure 11: FS-MS GA with S-LOC

Note: No.FF refers as number of fitness functions.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a hardware-accelerated parallel GA for fitness functions with variable execution time. We also introduce accelerated GA operators to further reduce the GA searching time. We compared our architecture with the state-of-the-art Master-Slave GA. The results shows that our architecture can achieve best performance (70% improvement) when the variability of fitness function execution time is significant, and that the accelerated GA operators can achieve higher improvement (99.57%) when the fitness function's computation time is long. The two improvement methods proposed in this paper can be assembled together in a single architecture, as presented here, or applied individually in the case of platforms with limited hardware resources, since they are not mutually dependent.

7. REFERENCES

[1] M. Yunfeng, L. S. Indrusiak, *Hardware-accelerated Response Time Analysis for Priority-Preemptive Networks-on-Chip*, in ReCoSoC 2015.

[2] M. Yunfeng, M. N. S. M. Sayuti, L. S. Indrusiak, *Inexact End-to-End Response Time Analysis as fitness function in search-based task allocation heuristics for hard real-time network-on-chips*, in ReCoSoC 2014.

[3] J.J. Grefenstette, *Parallel adaptive algorithms for function optimization*, in Technical Report No. CS-81-19, Vanderbilt University, 1981.

[4] S. M. Said and M. Nakamura, *Master-Slave Asynchronous Evolutionary Hybrid Algorithm and ITS Application in VANETs Routing Optimization*, in Advanced Applied Informatics (IIAIAAI), pp 960-965, 2014.

[5] D. Jakobovic, M. Golub, M. Cupic, *Asynchronous and implicitly parallel evolutionary computation models*, in Soft Comput, 2014.

[6] S. D. Scott, A. Samal, and S. Seth, *HGA: A hardware based genetic algorithm*, in Proc. ACM/SIGDA 3rd Int. Symp. FPGA's, pp.53–59, 1995.

[7] S. D. Scott, S. Seth and A. Samal, *A Hardware Engine for Genetic Algorithms*, in Technical Report UNL-CSE-97-001, University of Nebraska-Lincoln, 1997.

[8] C. Aporntewan and P. Chongstitvatana, *A hardware implementation of the compact genetic algorithm*, in Proc. IEEE Congr. Evol. Comput., vol. 1, pp. 624–629, 2001.

[9] P. Sen and P. Pateriya, *Implementation of generic algorithm using VHDL on FPGA*, in International Journal of Scientific & Engineering Research, vol. 2, 2011.

[10] K.Hsiue and B.Teague, *Hardware Implementation of Genetic Algorithms*, in Complex Digital Systems, 2013.

[11] R.Faraji and H.R.Naji, *An Efficient Crossover Architecture for Hardware Parallel Implementation of Genetic Algorithm*, in Neurocomputing, vol. 128, pp 316-327, 2014.

[12] L. S. Indrusiak, *End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration*, in Journal of Syst. Archit, 2014.